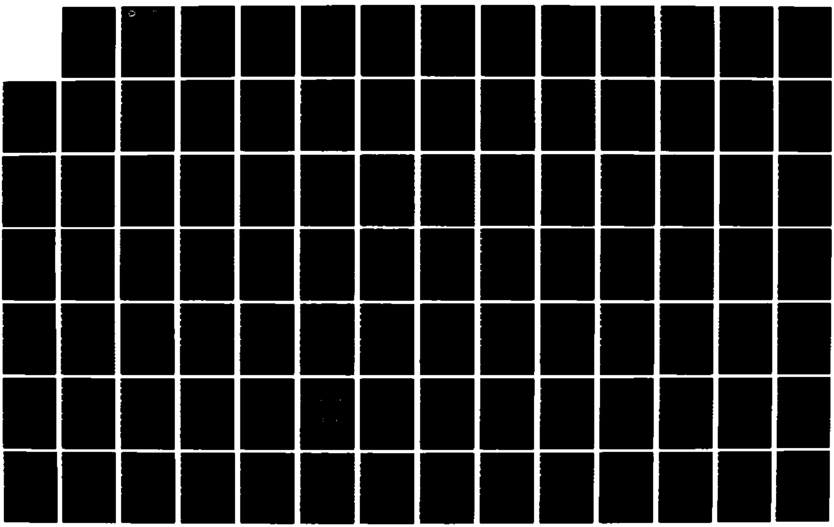
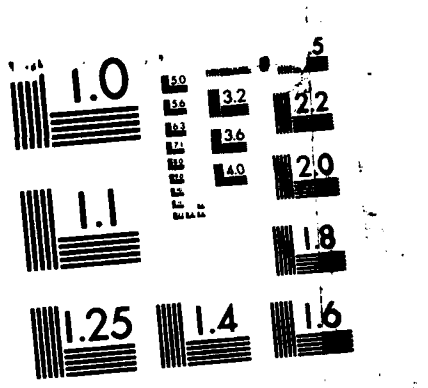
AD-A166 366 ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA PROGRAMMING L202 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 1/

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A166 366

1986

Ada® Training Curriculum

Basic Ada® Programming
L202
Teacher's Guide
Volume I

Superseded AD-A143584

U.S. Army Communications-Electronics Command
(CECOM)

Contract DAAB07-83-C-K506
14

Prepared By:

SOFTECH, INC.
460 Totten Pond Road
Waltham, MA 02154

86 4 2 042

®Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

PART I

BASIC Ada* PROGRAMMING (L202)

TEACHER'S GUIDE

VG 728.2

INSTRUCTOR NOTES

"THIS COURSE TEACHES A WORKING SUBSET OF THE Ada LANGUAGE. ITS INTENDED AUDIENCE IS BEGINNING Ada PROGRAMMERS WITH LITTLE EXPERIENCE WITH Ada. HOWEVER THIS COURSE DOES REQUIRE THE STUDENT TO HAVE WORKING KNOWLEDGE OF AT LEAST ONE HIGH ORDER PROGRAMMING LANGUAGE. THE COURSE PRESENTS THE FUNDAMENTAL CONCEPTS OF THE LANGUAGE. THIS COURSE DOES NOT ATTEMPT TO TEACH THE FULL Ada LANGUAGE."

BASIC Ada PROGRAMMING
Course Outline

INSTRUCTOR NOTES

KEEP SECTION 1 "LIGHT." DON'T WORRY ABOUT SPECIFIC POINTS. BREEZE THROUGH JUST TO SET THE STAGE FOR SECTION 2.

ALLOW 1 HOUR FOR SECTION 1.

THIS SECTION IS A REVIEW OF CONCEPTS COVERED IN M102, INTRODUCTION TO SOFTWARE ENGINEERING.

SECTION 1
INTRODUCTION TO SOFTWARE ENGINEERING

INSTRUCTOR NOTES

THE MAIN TOPICS TO BE DISCUSSED.

INTODUCTION TO SOFTWARE ENGINEERING

- OVERVIEW
- SOFTWARE ENGINEERING: GOALS AND PRINCIPLES
- ACHIEVING SOFTWARE ENGINEERING GOALS
  - LIFE CYCLE
  - REQUIREMENTS ANALYSIS
  - DESIGNING SOFTWARE
  - STRUCTURED PROGRAMMING
  - SUPPORTING SOFTWARE DEVELOPMENT
  - Ada AND SOFTWARE ENGINEERING

INSTRUCTOR NOTES

THIS IS A GOOD STOPPING PLACE TO ASK SOME OF THE STUDENTS, WHY ARE THEY TAKING THIS COURSE. IF THE REASONS ARE MANY, YOU COULD WRITE THEM ON THE VIEWGRAPH.

THE MAJOR REASON WILL PROBABLY BE:

USING Ada IS BEING REQUIRED OF ALL GOVERNMENT CONTRACTORS.

THE CONCEPTS ARE:

- ABSTRACTION
- INFORMATION HIDING
- MODULARITY
- ETC.

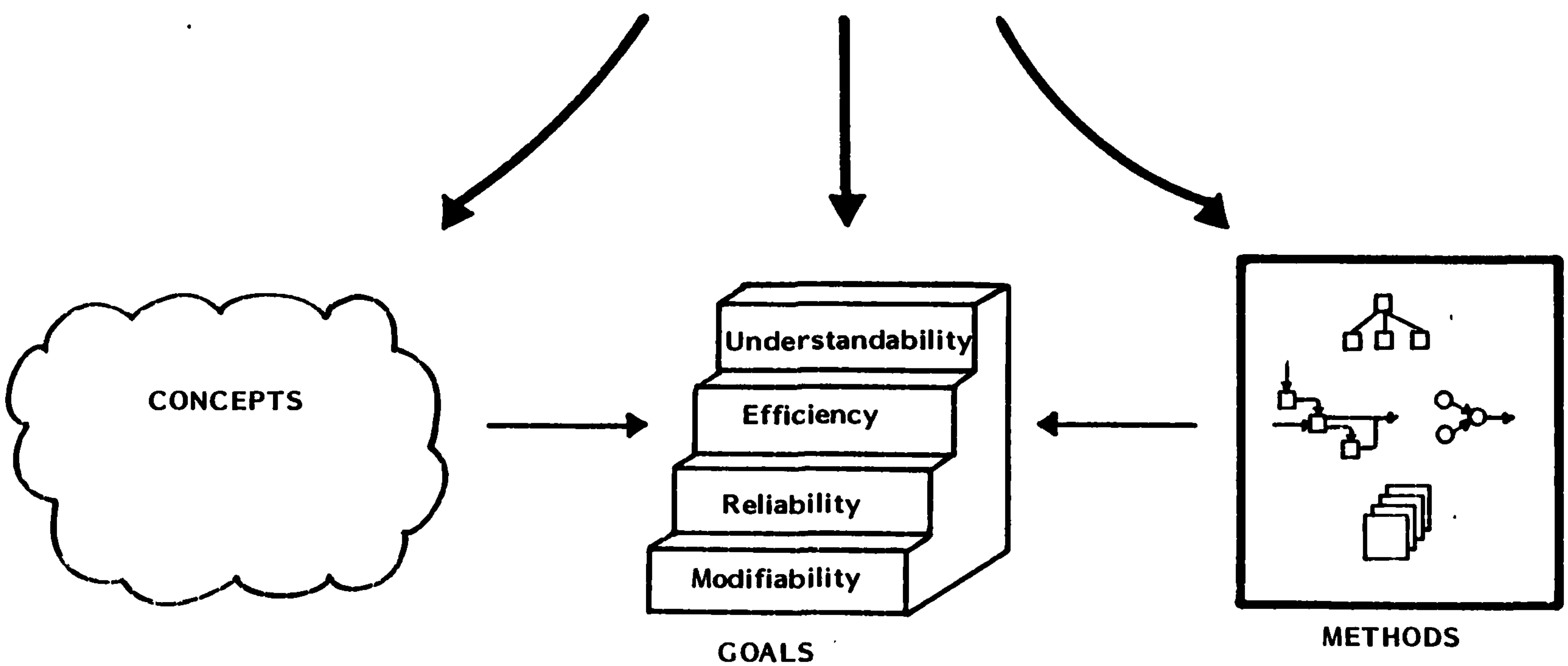
THE METHODS ARE:

- STRUCTURED DESIGN
- DATA FLOW DIAGRAMS
- ETC.

WHY AM I HERE?

Ada WAS DESIGNED TO SUPPORT

SOFTWARE ENGINEERING



- UNDERSTAND Ada BY FIRST UNDERSTANDING SOFTWARE ENGINEERING

INSTRUCTOR NOTES

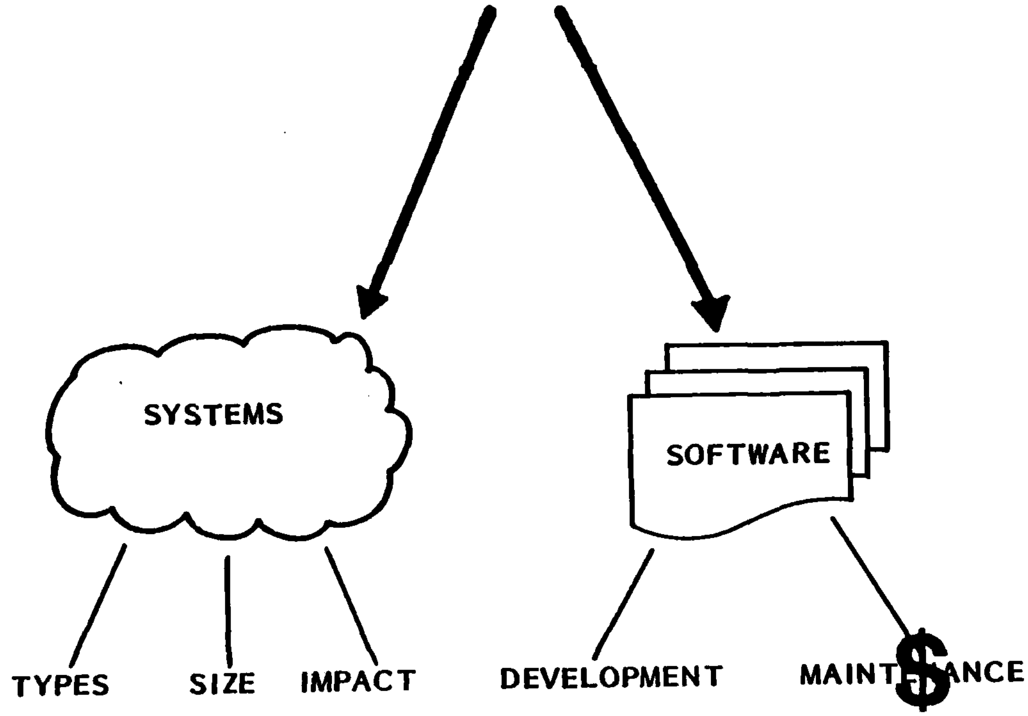
COMPUTER SYSTEMS HAVE AN IMPACT ON THEIR ENVIRONMENT, E.G. AN AIRLINE TRAFFIC CONTROL SYSTEM NEEDS TO BE MORE ROBUST WITH LOW FAILURE RATE AND HIGH AVAILABILITY RATE THAN A SYSTEM WHICH CUTS FABRIC FOR A CLOTHING FIRM. THE CLOTHING FIRM CAN AFFORD A SOMEWHAT HIGHER FAILURE RATE. THE IMPACT THAT THE FAILURE HAS ON ITS ENVIRONMENT IS THE POINT TO STRESS.

THE PROBLEM OF MAINTENANCE IS <u>STILL</u> HARD: IT'S VERY EXPENSIVE AND WE ARE STILL LEARNING.

THIS IS A FOUNDATION SLIDE. ADDRESS WHAT PROMPTED THE NEED FOR SOFTWARE ENGINEERING.

HISTORICAL PERSPECTIVE

SOFTWARE ENGINEERING IS A RESPONSE TO INCREASING COMPLEXITY



INSTRUCTOR NOTES

NO COMMON, SIMPLE DEFINITION FOR SOFTWARE ENGINEERING.

SEEWG = SOFTWARE ENGINEERING ENVIRONMENT WORKING GROUP, WORKING GROUP TO DEFINE REQUIREMENTS FOR A NAVY STANDARD SOFTWARE ENGINEERING ENVIRONMENT.

SOME DEFINITIONS

- "SOFTWARE ENGINEERING IS THE APPLICATION OF SCIENCE AND MATHEMATICS BY WHICH THE CAPABILITIES OF COMPUTER EQUIPMENT ARE MADE USEFUL TO MAN VIA COMPUTER PROGRAMS, PROCEDURES AND ASSOCIATED DOCUMENTATION."

  BOEHM 1981

- "... A SOFTWARE ENGINEERING PROCESS IS A SET OF ACTIVITIES FOR DEVELOPING AND MODIFYING SOFTWARE THROUGH ITS LIFE CYCLE"

  SEEWG REPORT 1982

- "A METHODOLOGY IS A REPEATABLE HUMAN PROCEDURE WHICH SUPPORTS SOME ASPECT OF AN ACTIVITY."

  SEEWG REPORT 1982

INSTRUCTOR NOTES

THIS IS THE DEFINITION OF THE SOFTWARE CRISIS THAT MOTIVATED THE DEVELOPMENT OF Ada.

GIVE SOME PERSONAL EXAMPLES OF SYSTEMS YOU HAVE WORKED ON OR ASK THE CLASS FOR SOME.

SOFTWARE ENGINEERING
(Software Crisis)

- SOFTWARE FOR COMPLEX MILITARY SYSTEMS
  - IS USUALLY LATE
  - COSTS MORE THAN ORIGINALLY ESTIMATED
  - DOES NOT WORK TO ORIGINAL SPECIFICATIONS
  - IS UNRELIABLE
  - IS DIFFICULT AND COSTLY TO MAINTAIN

INSTRUCTOR NOTES

TALK BRIEFLY TO EACH BULLET, GIVING THE CLASS PERSONAL EXPERIENCE OR TRY TO GET THEM TO RELATE THEIR EXPERIENCES.

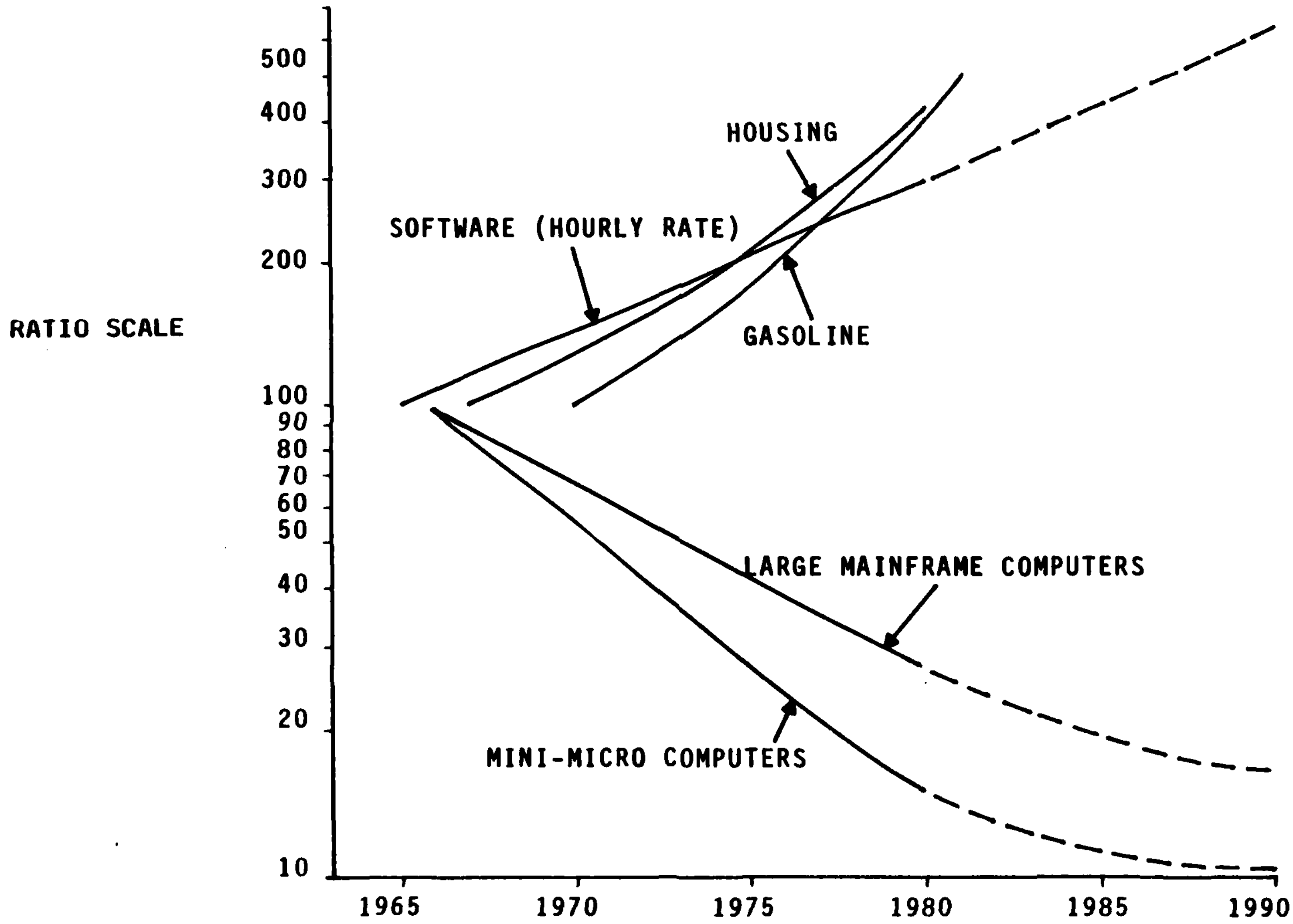
MOTIVATION FOR SOFTWARE ENGINEERING
(Additional Problems)

- SOFTWARE NOT REUSABLE ON DIFFERENT SYSTEMS
- PROLIFERATION OF METHODS, LANGUAGES AND ARCHITECTURES
- METHODS AND LANGUAGES NOT SUITED FOR CURRENT APPLICATIONS
- SUPPLY OF QUALITY SOFTWARE PERSONNEL NOT ABLE TO MEET CURRENT SOFTWARE DEMAND
- SOFTWARE TASKS MORE COMPLEX NOW, BUT NO WIDELY USED METHODS AND TOOLS TO DEAL WITH THE PROBLEM EXIST
- LACK OF ADEQUATE MANAGEMENT <u>AND</u> SOFTWARE DEVELOPMENT METHODS/TOOLS

INSTRUCTOR NOTES

NOTE THE DIRECTION OF THE VARIOUS COST FACTORS NOT THE ACTUAL VALUES.

SOFTWARE vs. HARDWARE COST TRENDS



SOURCE: REPORT OF THE SOFTWARE ENGINEERING GROUP, 1983

VG 728.2

INSTRUCTOR NOTES

BRIEFLY GO OVER EACH POINT. THIS SETS THE STAGE FOR THE STUDENTS.

POINT OUT THAT A WELL-CONCEIVED METHODOLOGY HAS CREATIVE, INTELLECTUAL, CLERICAL, AND MECHANICAL ASPECTS.

SOFTWARE ENGINEERING METHODOLOGIES

- GOOD SOFTWARE ENGINEERING METHODOLOGIES SHOULD
  - IMPROVE EFFECTIVENESS AND PRODUCTIVITY OF SOFTWARE DEVELOPMENT ACTIVITIES
  - RESULT IN THE CREATION OF RELIABLE SOFTWARE
  - FIT TOGETHER TO FORM AN INTEGRATED SET OF METHODS
  - SEPARATE THE CREATIVE ASPECTS FROM THE MECHANICAL ASPECTS
  - PROMOTE AUTOMATION OF THE CLERICAL ASPECTS OF SOFTWARE DEVELOPMENT

- BY APPLYING A SET OF METHODOLOGIES YOU WILL ACHIEVE HIGHER QUALITY SOFTWARE THAT FULFILLS THE NEEDS.

- INCREASED EFFORT IN THE EARLIER ACTIVITIES OF A DEVELOPMENT WILL BE REFLECTED IN REDUCED COSTS FOR TESTING AND MAINTENANCE
  - PREVENT ERRORS FROM BEING INTRODUCED
  - DETECT ANY ERRORS AT EARLIEST POSSIBLE TIME

INSTRUCTORS NOTES

KEEP THIS SECTION "LIGHT" ALSO.

- OVERVIEW
- SOFTWARE ENGINEERING: GOALS AND PRINCIPLES
- ACHIEVING SOFTWARE ENGINEERING GOALS
    - LIFE CYCLE
    - REQUIREMENTS ANALYSIS
    - DESIGNING SOFTWARE
    - STRUCTURED PROGRAMMING
    - SUPPORTING SOFTWARE DEVELOPMENT
    - Ada AND SOFTWARE ENGINEERING

INSTRUCTOR NOTES

THESE FUNDAMENTAL OBJECTIVES ARE COMMON TO ALL SYSTEM DEVELOPMENTS

- UNDERSTANDABILITY = LEGIBILITY
- UNDERSTANDABILITY = COMPREHENSION TO THE "RIGHT" INDIVIDUAL
- RELIABILITY CAN ONLY BE BUILT IN FROM THE START.
- MODIFIABILITY EASILY HANDLES EXPECTED CHANGE.
- OF ALL THESE, EFFICIENCY IS THE EASIEST TO TWEAK ("TUNING").

SOFTWARE ENGINEERING GOALS

- BUILD A SYSTEM THAT MEETS USER EXPECTATIONS
  - CORRECT
  - RELIABLE
  - UNDERSTANDABLE
  - TRACTABLE
  - VERIFIABLE
- BUILD A SYSTEM THAT CAN ACCOMMODATE CHANGE
  - MAINTAINABLE
  - MODIFIABLE
  - TRANSPORTABLE
  - REUSABLE
- BUILD A SYSTEM WITHIN AVAILABLE RESOURCES
  - EFFICIENT
  - PRODUCTIVE

INSTRUCTOR NOTES

TRADE-OFF MUST CONSIDER:

- WHICH GOAL IS MOST IMPORTANT TO THE USER?
- WHICH GOAL IS HARDEST TO ATTAIN OR RISKIEST?

AT THE START OF SYSTEM DEVELOPMENT, IT IS IMPORTANT TO ESTABLISH THE RELATIVE IMPORTANCE OF SYSTEM OBJECTIVES - I.E. IS THE ABILITY TO RESPOND TO CHANGE MORE IMPORTANT THAN GETTING IT RIGHT IN THE FIRST PLACE?

DON'T PLACE TOO MUCH EMPHASIS ON A GOAL THAT IS

- RELATIVELY UNIMPORTANT

  AND/OR

- RELATIVELY EASY

SOFTWARE ENGINEERING GOAL CONFLICTS

- SOME GOALS ARE MUTUALLY COMPATIBLE
  - UNDERSTANDABILITY, MODIFIABILITY
- CONFLICTING GOALS REQUIRE TRADE-OFFS
  - EFFICIENCY VS. UNDERSTANDABILITY
  - PRODUCTIVITY VS. EFFICIENCY
  - RELIABILITY VS. PRODUCTIVITY
- TRADE-OFFS BETWEEN GOALS BASED ON SYSTEM OBJECTIVES
- TRADE-OFFS MUST BE AGREED TO BY ALL PARTICIPANTS
  - USER, MANAGER, IMPLEMENTORS

INSTRUCTOR NOTES

NOT ALL ENGINEERING PRINCIPLES ADDRESS THE ENTIRE LIFE-CYCLE, SOME ARE LIMITED TO FRONT-END CONSIDERATIONS; OTHERS APPLY ONLY TO DESIGN AND IMPLEMENTATION.

SOME OF THESE PRINCIPLES ARE WIDELY ACCEPTED AND USED; OTHERS ARE STILL CONTROVERSIAL AND WHILE INTUITIVELY APPEALING, HAVE NOT BEEN WIDELY USED AND PROVEN "UNDER FIRE".

SOFTWARE ENGINEERING PRINCIPLES

- ARE THE TECHNIQUES FOR ATTAINING SOFTWARE GOALS
- APPLY TO DIFFERENT PHASES OF LIFE-CYCLE
- ARE USED IN MANAGING SYSTEM COMPLEXITY AND CHANGE, AND IMPROVING QUALITY
- REPRESENT AN EVOLVING CONSENSUS
  - NOT ALL PROVEN OR ACCEPTED THROUGHOUT INDUSTRY

INSTRUCTOR NOTES

QUESTION: WHO SPENDS MORE TIME READING A PROGRAM?

1) THE COMPUTER (TO EXECUTE IT)

OR

2) PEOPLE (TO REVIEW, DEBUG, MODIFY...)

ASK THE AUDIENCE!!

ANSWER: IT DEPENDS

BUT

THE AMOUNT OF HUMAN READING IS STAGGERING

SOFTWARE COMPLEXITY

SOME FACTORS:

- FUNCTIONALITY - MILLIONS OF LINES OF CODE
- DATA - LARGE DATABASES, WITH COMPLEX RELATIONSHIPS AMONG DATA ITEMS
- RELIABILITY - FAILURE CAN HAVE GRAVE CONSEQUENCES
- UNDERSTANDABILITY - HUNDREDS OF PERSON-YEARS SPENT BY PEOPLE READING THE CODE.
- MAINTAINABILITY - OPERATIONAL LIFETIME MUST BE COST-EFFECTIVE

INSTRUCTOR NOTES

THIS IS JUST AN EXAMPLE OF ONE POSSIBLE APPROACH. THIS IS NOT INTENDED TO BE A COOKBOOK SLIDE.

THESE APPROACHES ARE GENERAL, SPECIFICS ARE PROJECT RELATED.

DON'T GET DRAWN INTO A DISCUSSION ABOUT THE EXAMPLES.

MANAGING SOFTWARE COMPLEXITY

COMPLEXITY ISSUE	APPROACH	EXAMPLES
MILLION LINES OF CODE	BREAK INTO SMALL PIECES WITH WELL-DEFINED INTERFACES.	• PACKAGES • SUBROUTINES • STRUCTURED DESIGN
LARGE DATABASE	DEFINE THE CLASSES OF DATA AND THEIR RELATIONS.	• BACHMAN DIAGRAMS • RELATIONAL DBMS
RELIABILITY	SPECIFY AT REQTS ANALYSIS TIME.	• FAILSAFE SPECIFICATIONS
UNDERSTANDABILITY	EXPRESS ASPECTS OF THE SYSTEM FROM DIFFERENT VIEWPOINTS	• MULTIPLE MODELS IN VARIOUS LANGUAGES
MAINTAINABILITY	CONTROL AND COORDINATE DEVELOPMENT	• SOFTWARE ENGINEERING FACILITIES

INSTRUCTOR NOTES

HERE ARE THREE (3) WAYS TO HELP MANAGE CHANGE.

MUST CONSIDER CHANGE IN THE AREAS OF:

- TECHNOLOGY
- PROBLEM COMPLEXITY
- RELIABILITY NEEDS

MANAGING CHANGE

- STRICT CONFIGURATION CONTROL
  - ESTABLISH EARLY IN LIFE CYCLE
  - AUTOMATE TO THE HIGHEST EXTENT
- DESIGNING FOR EASE OF CHANGE
  - MODULAR DESIGN
  - HIGH COHESION
  - LOW COUPLING
- DOCUMENTATION

INSTRUCTOR NOTES

METRICS ARE THE MEASURING STICKS. THESE TECHNIQUES ALLOW US TO INSERT QUALITY WHEN USED PROPERLY.

IT IS NOT NECESSARY TO DISCUSS WHOLE SLIDE. PERHAPS JUST DISCUSS A COUPLE OF THE BULLETS. THE LIST IS FOR COMPLETENESS ONLY.

THESE METRICS ARE FROM THE CONSTANTINE METHODOLOGY.

IMPROVING QUALITY: METRICS

CONSTANTINE QUALITY METRICS ...

- COHESION - HOW WELL STATEMENTS INSIDE A MODULE COOPERATE WITH EACH OTHER
- COUPLING - THE WAY MODULES ARE CONNECTED TO EACH OTHER
- FAN-IN / FAN-OUT - THE NUMBER OF SUBROUTINES A MODULE HAS
- SCOPE-OF-EFFECT / SCOPE-OF-CONTROL - THE RANGE OF EFFECT OF A GIVEN DECISION POINT
- CONTROL STRUCTURE - THE AMOUNT OF COMMON SUPPORT MODULES LOW IN THE SYSTEM HIERARCHY

INSTRUCTOR NOTES

GO THROUGH EACH PRINCIPLE AND ITS BRIEF DEFINITION (MORE ON THE NEXT SLIDE).

SUPPRESSION OF DETAIL MEANS THAT YOU IDENTIFY AND OMIT -- FOR THE MOMENT -- NONESSENTIAL DETAILS.

THESE 3 PRINCIPLES ARE USED TOGETHER TO DEFINE ABSTRACT MODULE SPECIFICATIONS.

UNDERLYING PRINCIPLES

- MODULARITY: RATIONAL STRUCTURING
  - EACH MODULE HAS SPECIFIC FUNCTION WITH WELL-DEFINED INTERFACES
- ABSTRACTION: SUPPRESSION OF DETAIL
  - AIDS UNDERSTANDABILITY BY ALLOWING ESSENTIAL CHARACTERISTICS TO STAND OUT
- HIDING: MAKING SOME OF THE "HOW" UNKNOWN TO OTHER PARTS OF THE SYSTEM
  - DETAILS OF IMPLEMENTATION ARE HIDDEN, KEEPING INTERFACE WELL-DEFINED

INSTRUCTOR NOTES

EXAMPLES OF STRUCTURING: DIVIDING A LARGE SET OF OBJECTS (HARD TO COUNT) INTO A SMALLER SET OF GROUPS OF OBJECTS (EASIER TO COUNT)

UNDERLYING PRINCIPLES

- STRUCTURING: RELATING PARTS AND SUBPARTS
  - REDUCES COMPLEXITY; INCREASES UNDERSTANDABILITY
  - ESTABLISHES RULES & CONVENTIONS; REDUCES FLEXIBILITY

- SEPARATION OF CONCERNS: PHYSICALLY COLLECTING RELATED THINGS AND SEPARATING UNRELATED THINGS
  - REDUCES IMPACT OF CHANGE
  - ALWAYS KNOW WHERE TO FIND ANSWERS TO QUESTIONS

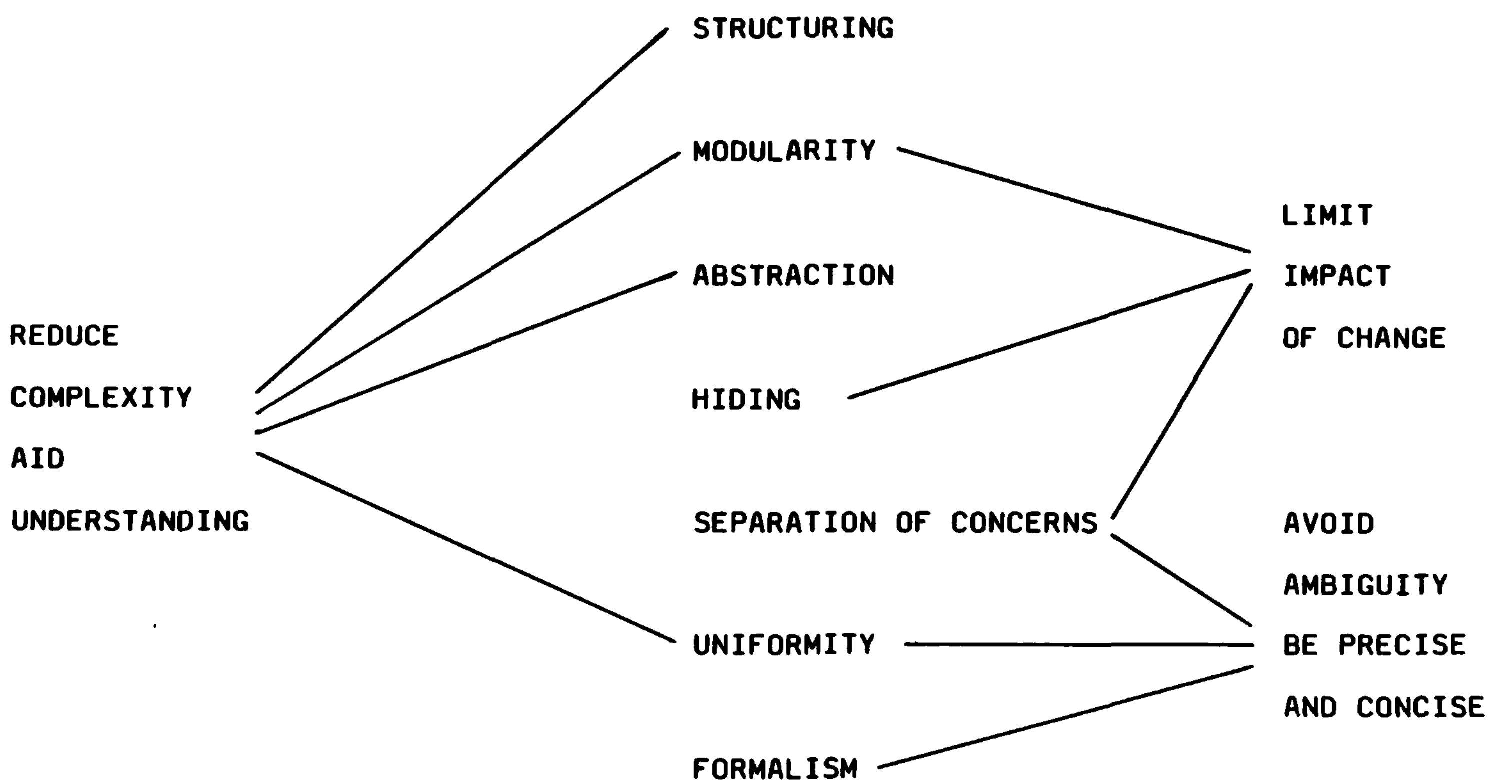
- UNIFORMITY: CONSISTENCY
  - INCLUDES NAMING CONVENTIONS, USE OF DATA TYPES AND DESIGN STRUCTURES

- FORMALISM: AVOIDING PROSE; BEING PRECISE/CONCISE
  - ALLOWS DEVELOPMENT OF DETAILED SPECIFICATION FOR REVIEW FOR COMPLETENESS/CORRECTNESS

INSTRUCTOR NOTES

THIS IS JUST A SUMMARY OF THESE PRINCIPLES WITH THE MOST IMPORTANT FOCUS OF THESE PRINCIPLES (IN TERMS OF OUR GOALS AND OBJECTIVES) POINTED OUT.

SOFTWARE ENGINEERING PROPOSALS SUMMARY



INSTRUCTOR NOTES

IF NEEDED, SPEND A LITTLE MORE TIME HERE.

- OVERVIEW
- SOFTWARE ENGINEERING: GOALS AND PRINCIPLES
- ACHIEVING SOFTWARE ENGINEERING GOALS
  - LIFE CYCLE
  - REQUIREMENTS ANALYSIS
  - DESIGNING SOFTWARE
  - STRUCTURED PROGRAMMING
  - SUPPORTING SOFTWARE DEVELOPMENT
  - Ada AND SOFTWARE ENGINEERING

INSTRUCTOR NOTES

BEFORE INVESTIGATING HOW TO ACHIEVE SOFTWARE ENGINEERING GOALS, WE NEED THE BIG PICTURE: LIFE-CYCLE.

THE CONCEPT OF A SOFTWARE DEVELOPMENT LIFE-CYCLE HAS EVOLVED OVER THE LAST 20 YEARS. IT IS VIEWED DIFFERENTLY BY DIFFERENT INDIVIDUALS, ORGANIZATIONS, AND INDUSTRIES, OFTEN BECAUSE OF THE TYPES OF PRODUCTS THAT ARE PRODUCED ARE RADICALLY DIFFERENT OR BECAUSE OF THE IMPORTANCE OR NON-IMPORTANCE OF THE SOFTWARE USED IN THE PRODUCT DEVELOPMENT.

SOFTWARE LIFE CYCLE

- LIFE CYCLE ORGANIZES THE ACTIVITIES OF BUILDING SOFTWARE
- SOFTWARE DEVELOPMENT IS BROKEN INTO PHASES
- LIFE CYCLE SUMMARIZES SOFTWARE DEVELOPMENT
- THE "LIFE CYCLE" IS NOT STANDARD THROUGHOUT THE INDUSTRY

INSTRUCTOR NOTES

TELL CLASS WE'LL NOW HIGHLIGHT (!!NO TIME FOR DETAILS!!) SOME IMPORTANT ASPECTS OF SOFTWARE DEVELOPMENT, KEEPING IN MIND THIS MODEL.

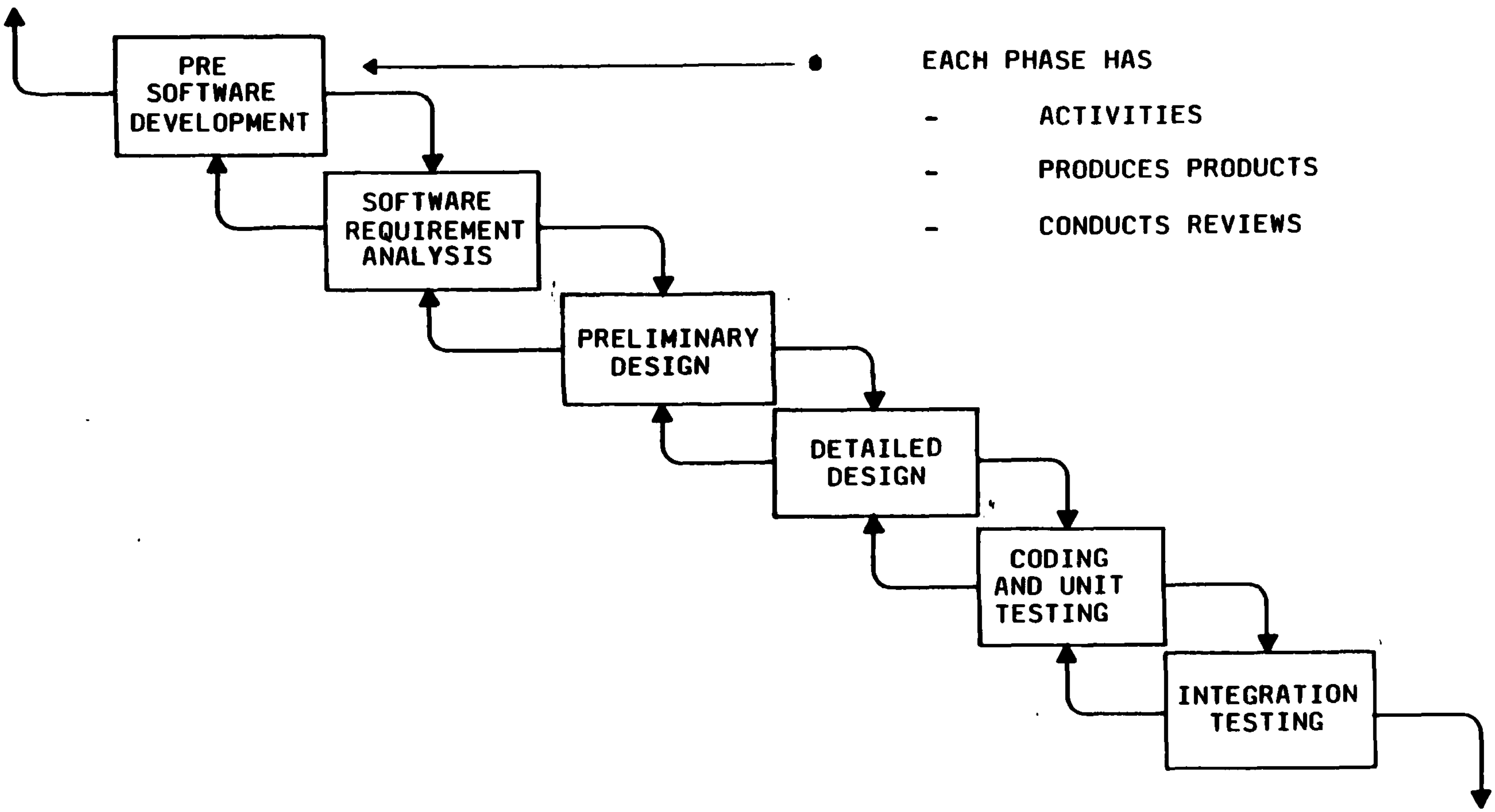
THIS IS REALLY DIVIDING UP THE RESPONSIBILITY OF THE SYSTEM DEVELOPMENT AMONG DIFFERENT GROUPS TO MAKE THE DEVELOPMENT PROCESS TRACTABLE. SOME DIVIDE IT UP INTO 6 PIECES, SOME INTO 4 OR 8 PIECES.

THIS PICTURE IS MORE CLASSICAL VIEW OF LIFE-CYCLE. THIS MAY (AND PROBABLY WILL) DIFFER FROM THE LIFE-CYCLE MODEL UNDERSTOOD BY EACH STUDENT. THIS IS OFTEN CALLED THE "WATERFALL" MODEL. OUT OF EACH PHASE A DELIVERABLE IS USUALLY PRODUCED, WHICH IS USED AS INPUT INTO THE NEXT PHASE. A REAL SYSTEM OR SOFTWARE DEVELOPMENT IS NEVER THIS CLEAN. THERE EXIST FEEDBACK LOOPS BETWEEN PHASES, THE DELIVERABLES NEED REVIEWS AND CHANGES BEFORE ACCEPTANCE, THE DESIGN PHASE MAY BE SUB-DIVIDED INTO MANY DESIGN PHASES OF INCREASED DETAIL, ETC.

THE KEY WORDS ARE:

- ANALYSIS - THE "WAIT" PHASE
- DESIGN - THE "HOW" PHASE
- IMPLEMENTATION - THE "BUILD" PHASE

DEVELOPMENT LIFE CYCLE MODEL



- EACH PHASE HAS
  - ACTIVITIES
  - PRODUCES PRODUCTS
  - CONDUCTS REVIEWS

INSTRUCTOR NOTES

WE ARE NOW GOING TO LOOK AT REQUIREMENTS AND DESIGN. THEY REALLY DON'T GET AS MUCH ATTENTION FROM THE DEVELOPERS AS THEY SHOULD.

REQUIREMENTS ANALYSIS IS USUALLY CONDUCTED WHILE AT THE SPECIFICATION AND/OR DESIGN LEVEL.

REQUIREMENTS ANALYSIS

- REQUIREMENTS ANALYSIS IS THE PROCESS BY WHICH THE FEASIBILITY OF REQUIREMENTS ARE DETERMINED, PRIOR TO THE DEVELOPMENT OF THE SYSTEM

- THE ANALYSIS IS TYPICALLY PERFORMED BY A SENIOR SYSTEMS EXPERT OR ANALYST WHO ESTABLISHES AND DOCUMENTS THE REQUIREMENTS

INSTRUCTOR NOTES

A REQUIREMENT IS A BINDING CONDITION WHICH STATES A MANDATORY CHARACTERISTIC OF AN ABSTRACT OR PHYSICAL OBJECT.

REQUIREMENTS MAY HAVE DIFFERENT FORMS: A SPECIFIC DESCRIPTION, A CONSTRAINT, AN EVALUATION CRITERIA FOR JUDGING QUALITY, OR IT MAY BE IMPLIED BY CONTEXT.

THERE'S ALWAYS MORE THAN ONE PARTY INVOLVED. ACHIEVING CONSENSUS IS AN IMPORTANT ASPECT FOR REQUIREMENTS ANALYSIS. STRESS THAT REQUIREMENTS ANALYSIS IS A HUMAN ENDEAVOR.

SPECIFYING REQUIREMENTS

- A REQUIREMENT IS ...
  - AN EXPRESSION OF <u>NEED</u>
  - AN IMPOSED <u>DEMAND</u>
  - SOMETHING SOMEONE WILL PAY FOR
- REQUIREMENTS FORM A "CONTRACT BETWEEN USER AND DEVELOPERS"
- REQUIREMENTS ARE DOCUMENTED IN REQUIREMENTS SPECIFICATION(S)

INSTRUCTOR NOTES

"WRONG" =

- MISUNDERSTOOD
- INCORRECTLY STATED
- FAILS TO CAPTURE REAL NEEDS
- EXPRESSES A BAD SOLUTION TO THE NEEDS

CONSEQUENCES OF WRONG REQUIREMENTS

WRONG REQUIREMENTS CAN CAUSE...

- SYSTEM REJECTION
- SYSTEM PATCHING OR RETROFIT
- INSTALLATION OF A DANGEROUS SYSTEM
- FAILURE OF THE PROJECT
- LOSS OF FUTURE BUSINESS

BUILDERS AND USERS MAY BE LOSERS!

INSTRUCTOR NOTES

GOOD ANALYSTS CAN EXPRESS THE <u>REAL</u> REQUIREMENTS, AS OPPOSED TO THE STATED OR PERCEIVED ONES. IT IS DIFFICULT JOB, AND MUST BE CAREFULLY THOUGHT OUT.

THE ANALYST IS THE BRIDGE BETWEEN USERS AND DEVELOPERS

- ANALYSTS ...:
  - POSTPONE DESIGN DECISIONS
  - PUT THEMSELVES IN THE USER'S CHAIR
  - QUESTION THE RATIONALE
  - UNDERSTAND THE <u>REAL</u> PROBLEM
  - CONSIDER SEVERAL SOLUTIONS
  - CLEARLY COMMUNICATE REQUIREMENTS AS WELL AS THE RESULTING IMPLEMENTATION

INSTRUCTOR NOTES

POINT OUT THAT DESIGNERS ALSO CONTRIBUTE TO THE ANALYSIS PHASE. THEY CITE PROBLEMS AND ERRORS FROM <u>THEIR</u> VIEWPOINT OF <u>FEASIBILITY</u>, AND CONVEY THIS TO THE ANALYST. ALSO STATE THAT THERE MAY BE MULTIPLE REQUIREMENT SPECIFICATIONS (SOFTWARE, HARDWARE, SYSTEM, ETC.) THAT NEED TO BE CREATED AND ITERATED. CONFLICTS AMONG THEM NEED TO BE SPOTTED BY THE ANALYST.

DESIGNER'S ROLE IN ANALYSIS

DURING ANALYSIS, DESIGNERS IDENTIFY:

- AMBIGUITIES
- INCONSISTENCIES
- INCOMPLETENESS
- POTENTIAL TROUBLE AREAS
- REQUIREMENTS HAVING DISPROPORTIONATE COST/SCHEDULE IMPACT

INSTRUCTOR NOTES

STRESS THAT REQUIREMENTS ANALYSIS IS A HUMAN ENDEAVOR.

ENGINEERS PREFER TALKING WITH MACHINES, NOT PEOPLE, SO ANALYSIS DOESN'T GET AS MUCH ATTENTION AS IT SHOULD.

TIPS

EXPECT ERRORS, BECAUSE PEOPLE...

- FORGET
- MISCOMMUNICATE
- CHANGE THEIR MINDS

DESIGN IS A BLUEPRINT OF MODULES AND THEIR INTERCONNECTIONS. THE DESIGN PROCESS ALLOCATES THE FUNCTIONAL REQUIREMENTS TO A DESIGN STRUCTURE, PRESENTING THE FORM OR DESIGN STRUCTURE ALONG WITH SOME INDICATION OF WHERE THE VARIOUS FUNCTIONS ARE TO BE PERFORMED. THE STRUCTURE AND ALLOCATION SHOULD ALLOW THE PERFORMANCE AND ANALYTIC REQUIREMENTS TO BE FACTORED IN AND VERIFIED AS CONSTRAINTS.

THE INTERFACES BETWEEN THE COMPONENTS OF THE STRUCTURE ARE ALSO DEFINED AT THIS TIME. (MORE EMPHASIS IS PLACED ON THESE INTERFACES (E.G. PARNAS) THAN OTHERS (E.G. STRUCTURED DESIGN)).

THE ACTIVITY OF DESIGN IS A <u>MODELING</u> ACTIVITY. A DESIGN LEAVES CERTAIN DETAILS OUT UNTIL LATER. THERE IS A NEED TO BE ABLE TO COMPREHEND A LARGE AMOUNT OF THE SYSTEM TO BE SURE THAT THE GOALS OF THE PARTICULAR DESIGN STRATEGY (WHICH VARY) ARE BEING MET.

DESIGNING SOFTWARE

- DESIGN <u>TRANSLATES</u> REQUIREMENTS SPECIFICATIONS INTO A <u>BLUEPRINT</u> OF THE SYSTEM.
- DESIGN IS A <u>MODEL</u> OF THE SOFTWARE.
- TWO MAJOR SUBPHASES
    - ARCHITECTURAL (PRELIMINARY) DESIGN
    - DETAILED DESIGN

INSTRUCTOR NOTES

DESIGNERS DEAL WITH MORE OF THE SYSTEM AT ONE TIME THAN IMPLEMENTERS. THEY CONCENTRATE ON GLOBAL ISSUES, POSTPONING DECISIONS HAVING ONLY LOCAL SCOPE. THE DESIGNER PARTITIONS THE SYSTEM IN A MANNER THAT HIDES DECISIONS LIKELY TO CHANGE SEPARATELY INTO INDIVIDUAL MODULES.

CLEAN INTERFACES REALLY MEAN FULLY DEFINED INTERFACES. THE SIMPLER THESE INTERFACES ARE - FEWER OPTIONS, FEWER PARAMETERS - THE MORE LIKELY IT IS TO BE FOR THEM TO BE FULLY DEFINED. COMPLEX INTERFACES PROBABLY MEAN TOO MANY FUNCTIONS IN A SINGLE MODULE.

THE DESIGNER COMMUNICATES WITH THE ANALYST EITHER VERBALLY OR BETTER YET, THROUGH DOCUMENTATION TO MAP FUNCTIONAL REQUIREMENTS (AS WELL AS OTHERS SUCH AS RELIABILITY, MODIFIABILITY, ETC.). THE DESIGNER WORKS WITH IMPLEMENTERS TO ASSESS PERFORMANCE AND OTHER GOALS IN ORDER TO MAKE TRADE-OFFS AMONG THEM.

THE INTERFACE BETWEEN DESIGNERS AND ANALYSTS IS PROBABLY MORE DIFFICULT THAN BETWEEN DESIGNERS AND IMPLEMENTERS. DESIGNERS USUALLY WERE IMPLEMENTERS ONCE AND UNDERSTAND THE AREA WELL.

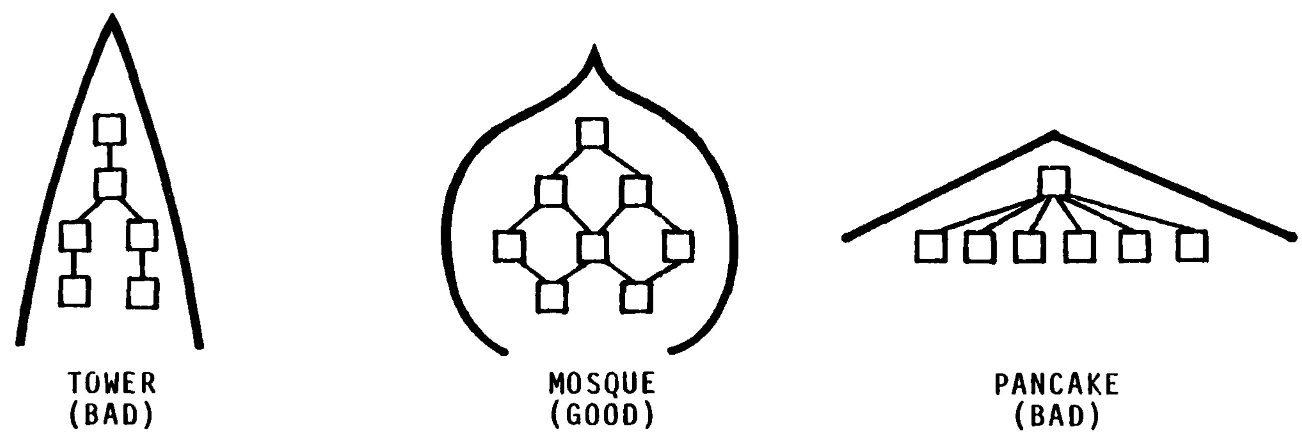
DESIGNER'S ROLE

- DESIGNERS ...
  - POSTPONE IMPLEMENTATION DECISIONS
  - HIDE THEIR DECISIONS INSIDE MODULES
  - WORRY ABOUT THE SOFTWARE'S STRUCTURE
  - SPECIFY ALGORITHMS AND CONTROL FLOW
  - MAKE CLEAN INTERFACES
  - COMMUNICATE WITH ANALYSTS AND IMPLEMENTERS

- THE DESIGNER IS THE BRIDGE BETWEEN ANALYSTS AND IMPLEMENTERS

INSTRUCTOR NOTES

IF YOU WANT TO HAVE A GOOD ARGUMENT ON REQUIREMENTS VERSUS DESIGN JUST VISIT AN ORGANIZATION WHERE ONE GROUP DOES ONE AND ANOTHER DOES THE OTHER. "WHAT" VS "HOW" SOUNDS SIMPLE BUT IT IS VERY HARD TO GET SPECIFIC ABOUT WHAT SOMETHING DOES WITHOUT PROPOSING A HOW. ADDING TO THE DIFFICULTIES IS THE FACT THAT PEOPLE GO FROM IMPLEMENTER TO DESIGNERS TO ANALYSTS. THEY HAVE A HARD TIME FORGETTING WHAT THEY PREVIOUSLY DID. THE DESIGN "BLUEPRINT" MUST DEPICT THE OVERALL STRUCTURE



SOME MEANS OF JUDGING OVERALL STRUCTURE IS IMPORTANT - GRAPHICS ARE NICE AND PREFERRED BUT OTHER METHODS ARE ACCEPTABLE (E.G., INDENTED HIERARCHAL DIAGRAM).

GENERAL GUIDELINES FOR ARHITECTURAL DESIGN

- THE BOUNDARY BETWEEN REQUIREMENTS AND DESIGN IS UNCLEAR (AND ALWAYS WILL BE)
  - IMPLIES ITERATION BETWEEN ANALYSIS AND DESIGN
- <u>NO SINGLE</u> METHOD GIVES A COMPLETE PROCESS ON HOW TO GO FROM REQUIREMENTS TO DESIGN
  - SOFTWARE COST REDUCTION PROJECT (SCRP) COMES CLOSE
- ALWAYS BUILD A PICTURE OF THE SOFTWARE ARCHITECTURE
  - MAKE ARCHITECTURE VISIBLE
  - SHOW INTERRELATIONSHIP BETWEEN COMPONENTS THAT MAKE UP ARCHITECTURE
- MAKE YOUR GOAL A SET OF DELIVERABLES

INSTRUCTOR NOTES

WE ARE REALLY TOUCHING A LITTLE OF THE IMPLEMENTATION PHASE.

STRUCTURED PROGRAMMING IS MORE A TECHNIQUE FOR WRITING GOOD CODE. IMPLEMENTATION IS MORE ATTUNED TO THE PRODUCTION OF THE CODE (SOFTWARE SYSTEM) WITHIN THE PROJECT SETTING.

POINT OUT THAT THERE IS NO ONE DEFINITION AGREED UPON BY EVERYONE.

STRUCTURED PROGRAMMING

- PROGRAMMING IS A MODELING ACTIVITY AND MODELS MUST BE REVIEWED BY PEOPLE ...
  - TO VERIFY CORRECTNESS OF THE APPROACH
  - TO FIND ERRORS
  - TO SHARE TECHNIQUES
- PROGRAMS MUST BE DESIGNED AND IMPLEMENTED TO BE <u>READ</u> AND <u>UNDERSTOOD</u> BY <u>PEOPLE</u>, NOT JUST COMPUTERS.
- STRUCTURED PROGRAMMING IS A COLLECTION OF TECHNIQUES WHICH EVOLVED; IT IS CONCERNED WITH THE PROGRAMS PEOPLE MAKE.

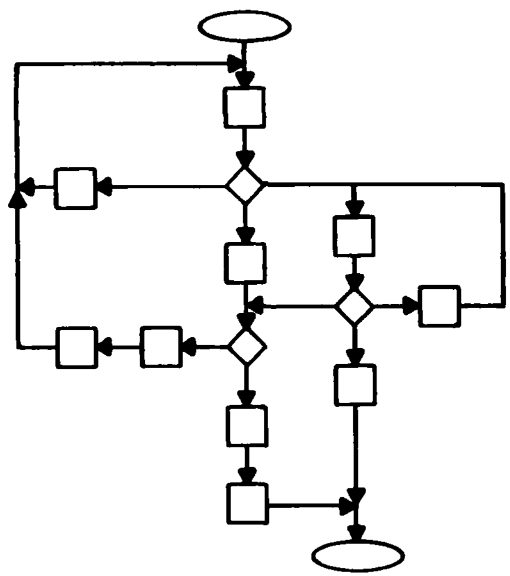
INSTRUCTOR NOTES

- SOFTWARE IS ALWAYS READ MORE OFTEN THAN IT IS WRITTEN.
- EARLY PROGRAMS WERE JUST WRITTEN, WITH NO THOUGHT OF HUMAN READER CONCERNS.
- STRUCTURED PROGRAMMING KEEPS THINGS SIMPLE, THUS ENHANCING THE LIKELIHOOD THAT THINGS WILL BE CORRECT.
- THE LEFT-HAND DIAGRAM REQUIRES A GREAT AMOUNT OF WORK TO DETERMINE WHAT CONDITION HOLDS AT A SPECIFIC POINT. MOREOVER, IT'S NOT CLEAR WHERE THAT POINT EXISTS.
- POINT OUT THAT STRUCTURED PROGRAMMING IS A METHOD, WITH RULES/GUIDELINES FOR PROGRAM STRUCTURE.

STRUCTURING RULES:

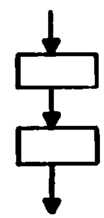
FLOW DIAGRAM OF

UNSTRUCTURED PROGRAM:

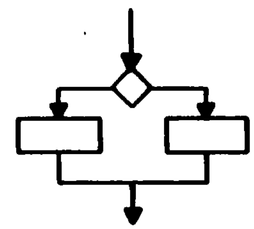


VERSUS

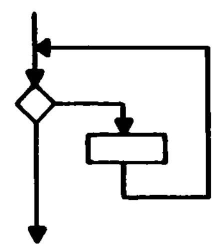
RULES FOR "STRUCTURING" PROGRAMS:



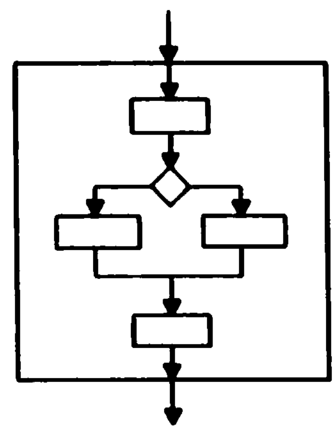
SEQUENCE



DECISION



REPETITION



HIERARCHY

INSTRUCTOR NOTES

TOP-DOWN

NOTE: SUBUNITS CAN BE REUSED

BOTTOM-UP

METHODS

PROGRAMS CAN BE BUILT...

- TOP-DOWN
  - USE STUBS/SUBUNITS
  - REQUIRES THAT NO SURPRISES OCCUR AT LOWER LEVELS
- BOTTOM-UP
  - USE "LIBRARY" OF SUBPROGRAMS/PACKAGES IMPORTED BY "DRIVER" UNIT(S)
  - VERIFIES LOW-LEVEL DESIGN ASSUMPTIONS
  - REQUIRES EXTRA CODE (THE DRIVERS) TO SIMULATE THE SYSTEM

INSTRUCTOR NOTES

AN OBSERVATION: COMPUTER PEOPLE ARE NOTORIOUS FOR NOT USING THE COMPUTER TO HELP THEM.

EMPHASIZE THAT AN ENVIRONMENT CONSISTS OF AN INTEGRATED SET OF TOOLS USED TO SUPPORT SOFTWARE DEVELOPMENT.

SOFTWARE DEVELOPMENT ENVIRONMENTS

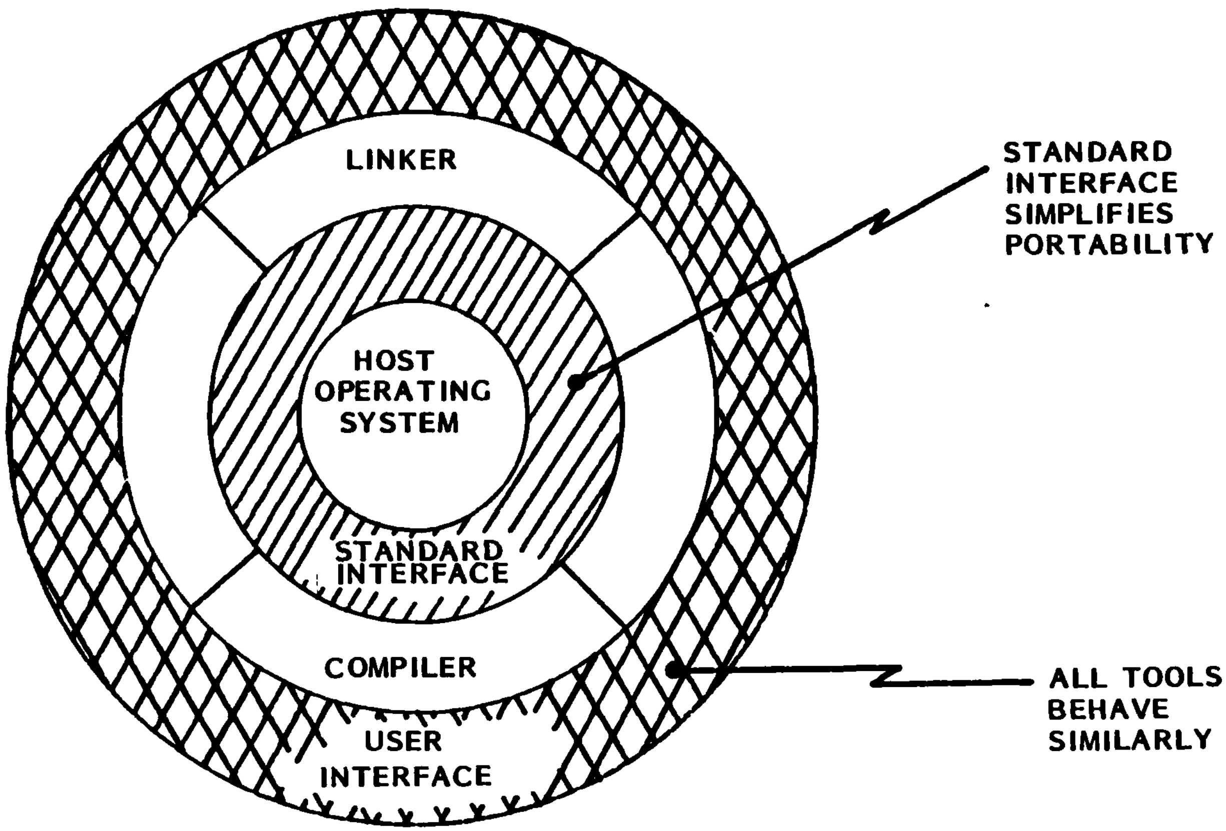
- AID EVERYONE ON THE TEAM
- ARE SETS OF INTEGRATED TOOLS
- KEEP EVERYONE'S INDIVIDUAL WORK SEPARATE
- CONTROL THE INTEGRATION OF EVERYONE'S WORK

INSTRUCTOR NOTES

THIS IS NOT AN EXACT PICTURE OF AN APSE ARCHITECTURE; JUST A SUMMARY TO CONVEY A MESSAGE.

THE POINT IS THAT A GOOD SOFTWARE DEVELOPMENT ENVIRONMENT NEEDS TO SEPARATE THE TOOLS FROM THE MACHINE: THE TOOLS NEED TO BEHAVE IN SIMILAR WAYS.

TOOLS



THE Ada APSE STRATEGY INTEGRATES TOOLS SO THEY:

- CAN BE PORTED
- WORK IN SIMILAR WAYS

INSTRUCTOR NOTES

CONFIGURATION MANAGEMENT CONSISTS OF NAMING (LABELING) THE ENTITIES OF THE SYSTEM, TRACKING THE ENTITIES THROUGH THE SYSTEM, AND CONTROLLING AND MANAGING CHANGES TO THE ENTITIES.

CONFIGURATION MANAGEMENT

- A CM SYSTEM CONSISTS OF

  - A SET OF DOCUMENTED MANUAL PROCEDURES
  - PROCEDURES AUTOMATICALLY ENFORCED
  - AUTOMATED TOOLS

  TO SUPPORT CONFIGURATION MANAGEMENT FUNCTIONS.

INSTRUCTOR NOTES

NOTE THAT NOT ALL OF THESE FEATURES WOULD BE AUTOMATED IN MOST CM SYSTEMS ALTHOUGH FUTURE SYSTEMS WILL PROBABLY ACCOMPLISH THIS AUTOMATION. CURRENTLY VERY FEW SYSTEMS AUTOMATICALLY PERFORM TESTING AND RETESTING OF COMPONENTS - FASP (FACILITY FOR AUTOMATED SOFTWARE PRODUCTION) ON NADC (NAVAL AIR DEVELOPMENT CENTER IN WARMINSTER, PA) ENVIRONMENT DOES PROVIDE THIS LAST CAPABILITY. TO BE COMPLETELY EFFECTIVE, ALL OF THESE CAPABILITIES SHOULD BE INTEGRATED, AS WELL AS AUTOMATED.

CM SYSTEM (Continued)

- THE BASIC LEVEL OF SUPPORT INCLUDES:
  - PROGRAM LIBRARIES (COLLECTIONS)
  - PROBLEM/CHANGE REPORTING PROCEDURES AND FORMS (OUTSIDE CHANGES)
  - ERROR TRACKING TOOLS AND PROCEDURES (BUG FIXES)
  - DOCUMENTATION CONTROL TOOLS AND PROCEDURES (CONSISTENT CHANGES)
  - AUTOMATIC TESTING AND RETESTING OF COMPONENTS

INSTRUCTOR NOTES

TO REACH THE SOFTWARE ENGINEERING GOALS.

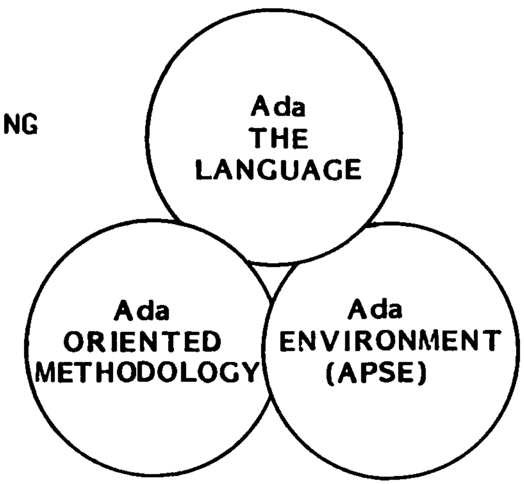
- HAS LANGUAGE FEATURES THAT SUPPORT SOFTWARE ENGINEERING TECHNIQUES.
- HAS AN ENVIRONMENT THAT CAN BE PORTED AND CAN HELP MANAGE CONFIGURATION ISSUES & SUPPORT DIFFERENT TEAM ORGANIZATIONS.

Ada AND SOFTWARE ENGINEERING ARE LIKE SOFTWARE AND HARDWARE - INSEPARABLE.

Ada AND SOFTWARE ENGINEERING RELATIONSHIP OF "Ada THE LANGUAGE" AND METHODOLOGIES

- ORIENTED TO OUR APPLICATIONS
- MANAGES THE COMPLEXITY ASSOCIATED WITH SOFTWARE SYSTEMS

- REDUCE COST OF DEVELOPING SYSTEMS
- INCREASE PORTABILITY OF SOFTWARE AND DEVELOPERS



- IMPROVE PRODUCTIVITY
- INCREASE SOFTWARE RELIABILITY, MAINTAINABILITY

THREE ASPECTS WHICH ADDRESS THE "SOFTWARE PROBLEM"

INSTRUCTOR NOTES

THIS SLIDE IS INTENDED TO BE VERY HIGH-LEVEL AND IS BY NO MEANS COMPLETE. THE PURPOSE IS TO GIVE STUDENTS A GENERAL IDEA OF THE INTEGRAL RELATIONSHIP BETWEEN Ada AND SOFTWARE ENGINEERING. SUBPROGRAMS WERE OMITTED BECAUSE THEY ARE PRESENT IN ALMOST ALL LANGUAGES AND ARE KNOWN TO SUPPORT MODULARITY AND PROGRAM COMPLEXITY MANAGEMENT. IN DISCUSSING THIS SLIDE, DO NOT DESCRIBE IN DEPTH WHAT THE Ada FEATURES MEAN; THEY ARE COVERED IN THE COURSE, SOME AT AN OVERVIEW LEVEL (GENERICS, PRIVATE TYPES, TASKS), OTHERS IN DEPTH.

THE CONTENTS OF THE TABLE DESCRIBE HOW THE FEATURE SUPPORTS THE CONCEPT. FOR INSTANCE, DATA TYPES ARE A MECHANISM TO REALIZE ABSTRACTION.

ASSIGN CHAPTER 1 OF THE PRIMER.

SUMMARY: Ada FEATURES SUPPORTING SOFTWARE ENGINEERING PRINCIPLES

	MODULARITY	ABSTRACTION	INFORMATION HIDING
PACKAGES	PROGRAM COMPILATION UNIT	MECHANISM	NAME SPACE CONTROL SPECIFICATION VERSUS BODY
DATA TYPES		ABSTRACT MODEL	
PRIVATE TYPES		MECHANISM	SEPARATION OF CONCERNS DATA ENCAPSULATION
GENERICS	REUSABILITY	REUSABILITY	
SEPARATE COMPILATION	MANAGING PROGRAM DEVELOPMENT	SPECIFICATION VERSUS BODY	
TASKS	COROUTINES, CONCURRENT PROCESSING	ELEGANT DESIGN (E.G. MONITOR)	

INSTRUCTOR NOTES

THIS SECTION IS A REVIEW OF MATERIAL WHICH APPEARS IN BOTH L101 AND L102.

SECTION 2
Ada TECHNICAL OVERVIEW

INSTRUCTOR NOTES

THIS SECTION SETS THE HISTORICAL MOTIVATION FOR Ada AND OUTLINES ITS DEVELOPMENT HISTORY.

TOPIC OUTLINE

BACKGROUND AND RATIONALE FOR Ada

WRITING AN Ada PROGRAM FROM BEGINNING TO END

SUMMARY OF Ada FEATURES

INSTRUCTOR NOTES

A LIST THAT CHARACTERIZES THE PRESENT STATE OF SOFTWARE DEVELOPED FOR EMBEDDED COMPUTER SYSTEMS.

ALTHOUGH IT'S CUSTOMARY TO BLAME PROGRAMMERS FOR ALL THIS, THE WHOLE "SYSTEM" IS TO BLAME.

- MANAGERS
- PROCUREMENT PRACTICES

  ...

  ...

SOFTWARE CRISIS: MOTIVATION FOR Ada

SOFTWARE FOR COMPLEX MILITARY SYSTEMS

- IS USUALLY LATE
- COSTS MORE THAN ORIGINALLY ESTIMATED
- DOES NOT WORK TO ORIGINAL SPECIFICATIONS
- IS UNRELIABLE
- IS DIFFICULT AND COSTLY TO MAINTAIN

INSTRUCTOR NOTES

FOLLOWING ARE SEVERAL GRAPHS AND A LIST OF UNDERLYING PROBLEMS ASSOCIATED WITH THIS"SOFTWARE CRISIS"

BRIEFLY GO THROUGH THESE.

PROBLEMS ASSOCIATED WITH THE SOFTWARE CRISIS

INSTRUCTOR NOTES

IN 1965, COST OF DEVELOPING A SOFTWARE SYSTEM WAS PRIMARILY A HARDWARE COST.

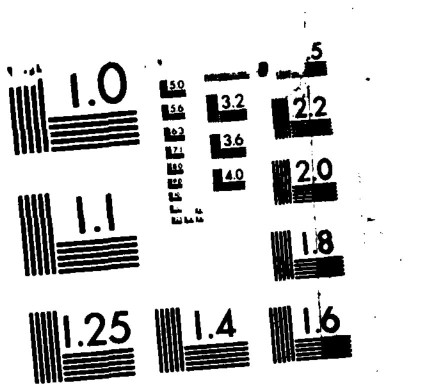
AROUND 1970 THIS BREAKDOWN OF TOTAL COST OF A SYSTEM WAS SPLIT FAIRLY EVENLY BETWEEN HARDWARE AND SOFTWARE.

BUT SINCE THEN, SOFTWARE COSTS FOR A SYSTEM HAVE RISEN DRAMATICALLY WHILE HARDWARE COSTS HAVE PLUMMETED AS A RESULT OF MICRO-CHIP TECHNOLOGICAL ADVANCES.

SOURCE: BARRY BOEHM, DEC 1976 IEEE TRANSACTIONS.

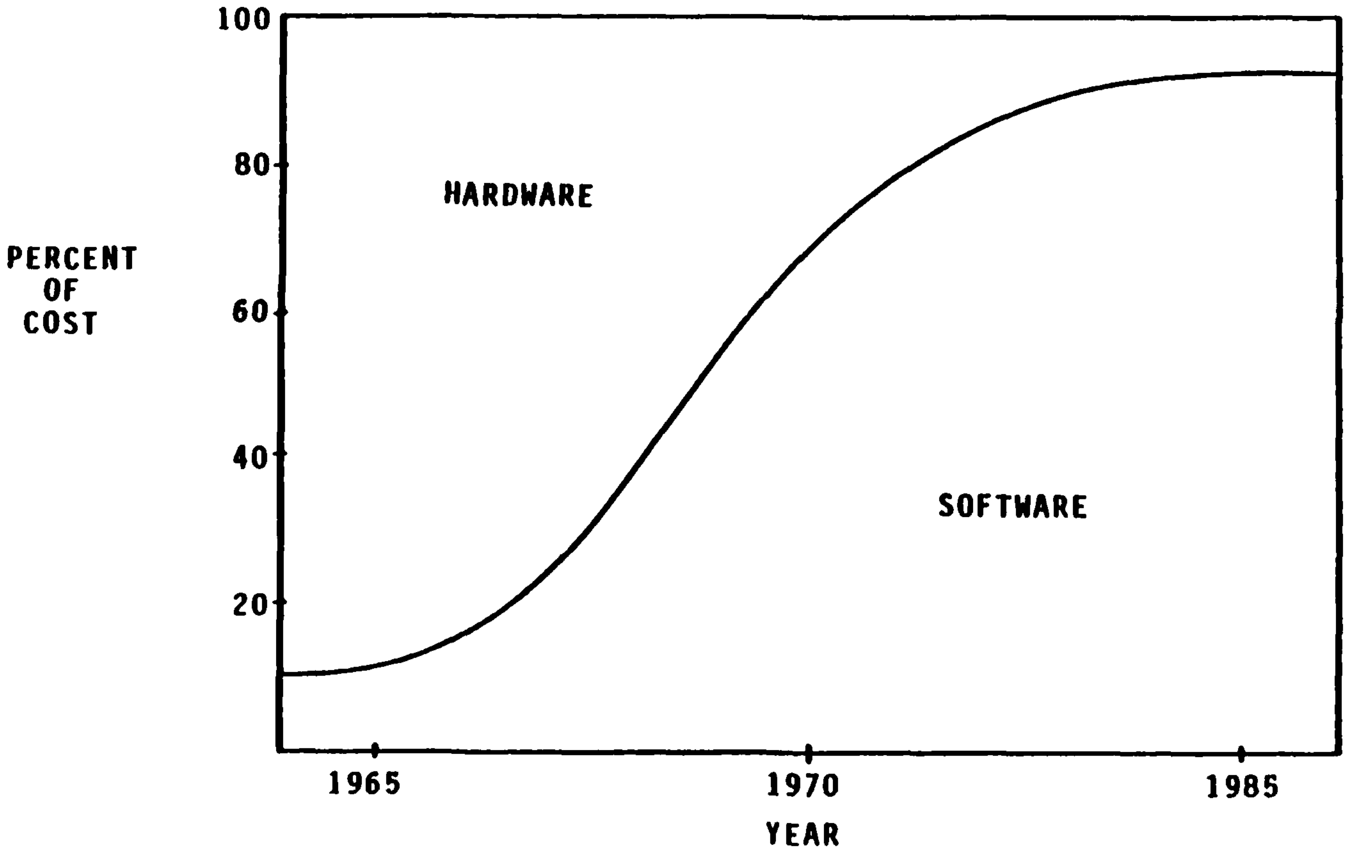
AD-A166 366 ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA PROGRAMMING L202 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 2/8

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

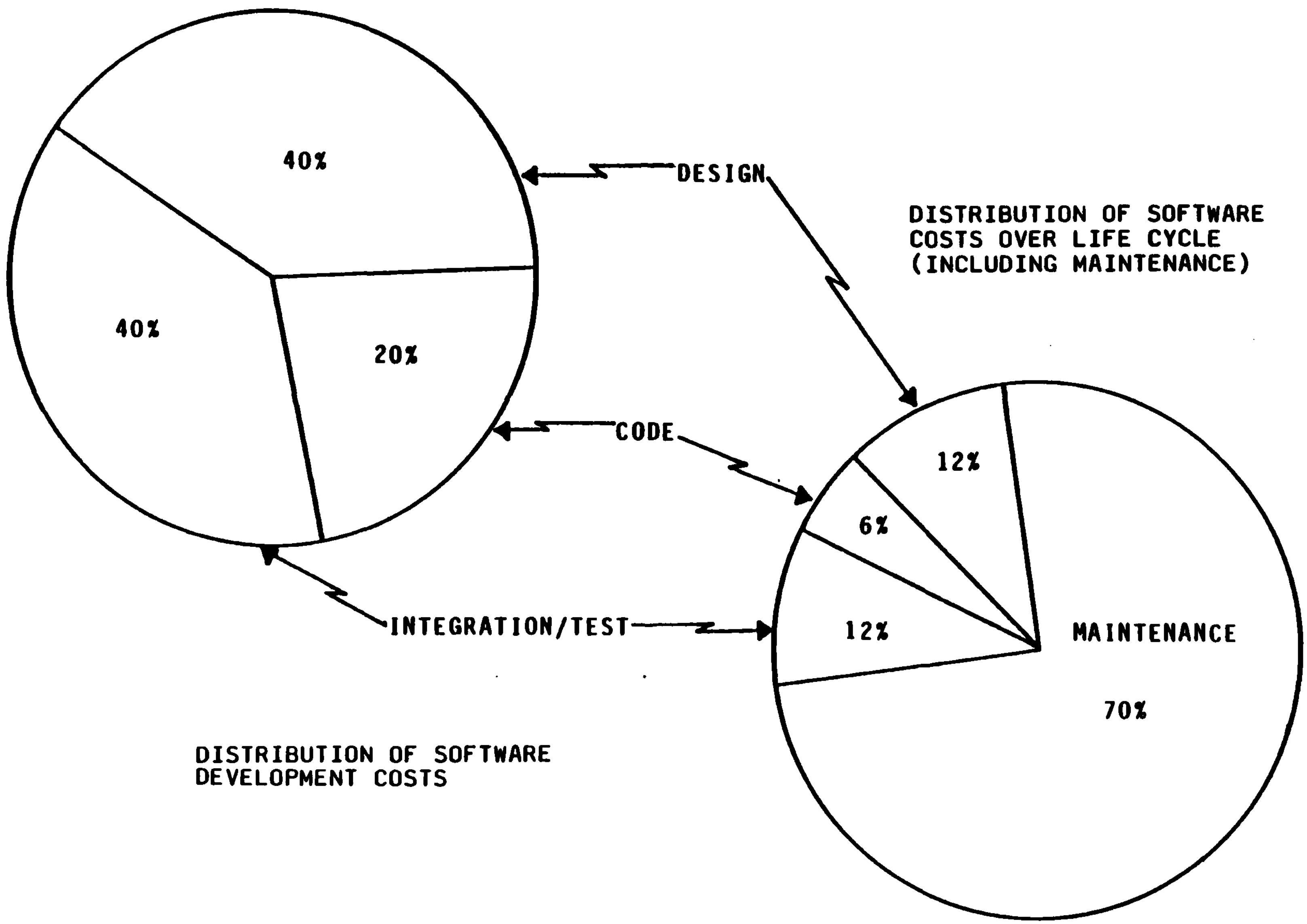
SOFTWARE COSTS INCREASE AS HARDWARE COSTS DECREASE



INSTRUCTOR NOTES

THE CAUSE OF THE INCREASED SOFTWARE COSTS IS THE SPECIFIC COST OF MAINTAINING/UPGRADING A SYSTEM ONCE IT IS OPERATIONAL.

SOFTWARE MAINTENANCE NEARLY TRIPLES ORIGINAL DEVELOPMENT COSTS



VG 728.2

INSTRUCTOR NOTES

AN ADDITIONAL COST WITH SOFTWARE LIES IN ERROR DETECTION AND CORRECTION.

FOR EXAMPLE:

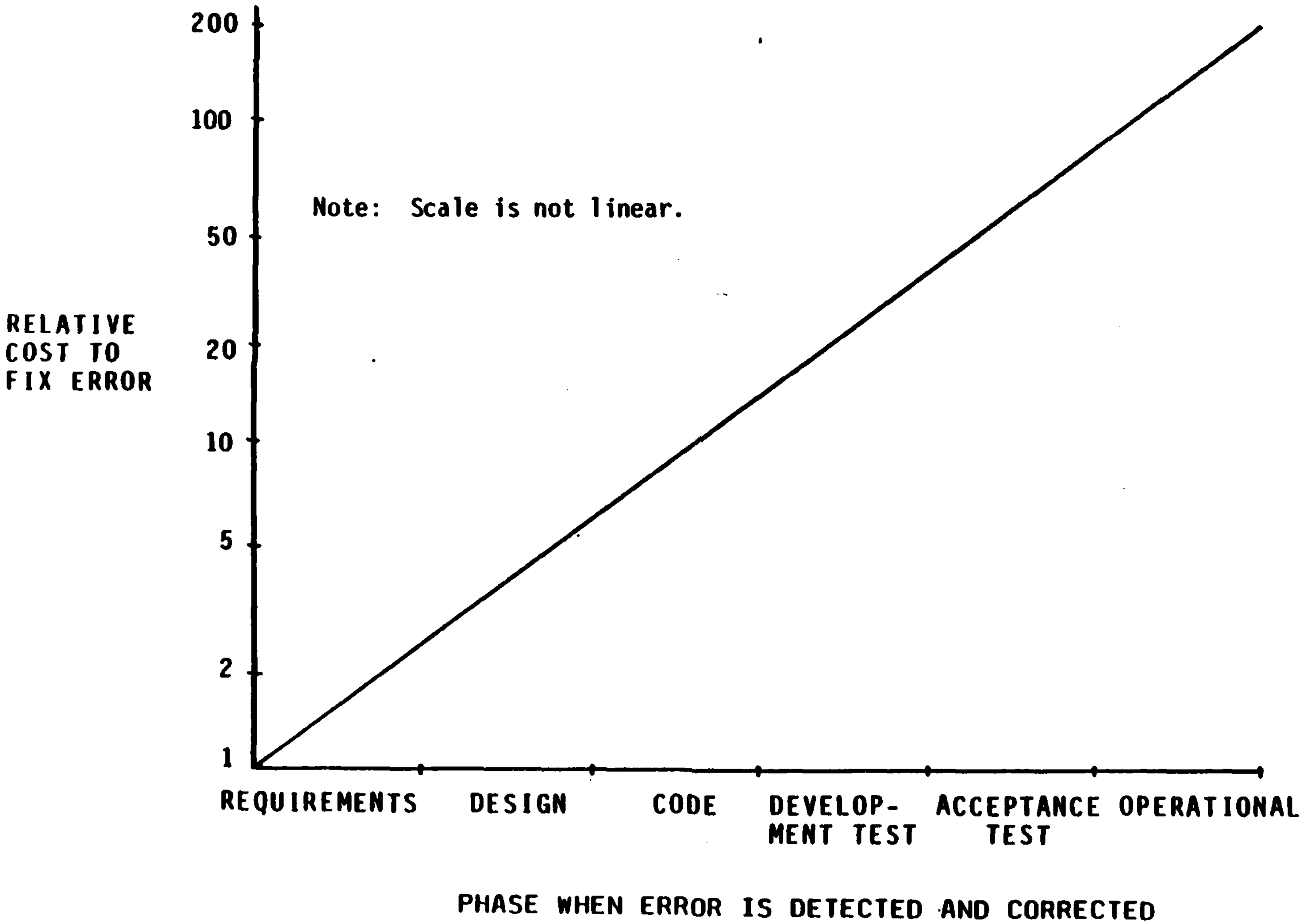
IF A REQUIREMENTS ERROR IS FOUND AND CORRECTED DURING THE REQUIREMENTS PHASE, YOU CAN JUST CORRECT THE REQUIREMENTS DOCUMENT WITH LITTLE COST IMPACT OF THE ERROR.

IF THE SAME ERROR IS NOT FOUND AND CORRECTED UNTIL MAINTENANCE, THE CORRECTION INVOLVES NOT ONLY DOCUMENT CHANGES SUCH AS SPECIFICATIONS, USER MANUALS, TRAINING MANUALS, BUT ALSO WILL INVOLVE VARIOUS AMOUNTS OF CODE MODIFICATIONS AND REVALIDATION. ERROR CORRECTION AT THIS POINT IN THE LIFE CYCLE IS TYPICALLY 100 TIMES WHAT IT WOULD HAVE BEEN IN THE REQUIREMENTS PHASE.

SOURCE: B. BOEHM, SOFTWARE ENGINEERING ECONOMICS, 1981

DATA IS FROM STUDIES BY IBM, TRW, GTE ON THIS TOPIC

COST OF ERROR CORRECTION



INSTRUCTOR NOTES

OTHER ASSOCIATED PROBLEMS WITH DECREASED PRODUCTIVITY AND RELIABILITY OF OUR SOFTWARE ARE THAT THE PROBLEMS WE ARE ATTEMPTING TO SOLVE NOW ARE MUCH MORE COMPLEX THAN IN THE PAST. COMPLEXITY ALONE IS NOT A PROBLEM, IT'S THE LACK OF ADEQUATE TOOLS TO ASSIST.

ADDITIONAL PROBLEMS

- SOFTWARE TASKS ARE MORE COMPLEX NOW, BUT NO ADEQUATE TOOLS TO DEAL WITH THE PROBLEM EXIST
- SUPPORT TOOLS (ASSEMBLERS, LINKERS, DEBUGGER) FOR EACH MACHINE ARE DIFFERENT
- LACK OF ADEQUATE MANAGEMENT AND SOFTWARE DEVELOPMENT TOOLS

INSTRUCTOR NOTES

AS ARCHITECTURES HAVE PROLIFERATED, SO TOO HAVE LANGUAGES. PLUS THE SUPPORT TOOLS FOR EACH ARCHITECTURE/LANGUAGE COMBINATION MUST BE DEVELOPED ANEW. OUR CURRENT LANGUAGES ARE NOT WELL SUITED TO THE NEEDS OF EMBEDDED COMPUTER SYSTEMS.

ADDITIONAL PROBLEMS

- SOFTWARE IS NOT REUSABLE ON DIFFERENT SYSTEMS
- PROLIFERATION OF LANGUAGES AND ARCHITECTURES
- LANGUAGES NOT SUITED FOR CURRENT APPLICATIONS
- SUPPLY OF QUALITY SOFTWARE PERSONNEL NOT ABLE TO MEET CURRENT SOFTWARE DEMAND.

INSTRUCTOR NOTES

IT IS A RETHINKING OF THE WAY IN WHICH SOFTWARE SYSTEMS WILL BE DEVELOPED IN THE FUTURE, WITH THE ITEMS LISTED AS VEHICLES OF THAT CHANGE. NOTE THAT IT IS THE COMBINATION OF LANGUAGE, ENVIRONMENT, AND METHODOLOGIES THAT CONSTITUTES THE Ada EFFORT.

WHEN WE SPEAK OF MODERN SOFTWARE ENGINEERING METHODS, WE ARE REFERRING TO SUCH THINGS AS STRUCTURED DESIGN AND PROGRAMMING, TOP-DOWN DEVELOPMENT, STRONG DATA TYPING, MODULARITY.

"RELIABLE SOFTWARE" IMPLIES THAT THE SOFTWARE PRODUCT CAN DETECT AND POSSIBLY RECOVER FROM ERROR OR FAILURE CONDITIONS IN OPERATION AND THAT SPECIAL MEASURES HAVE BEEN TAKEN TO PREVENT ERRORS IN AN ANALYSIS, DESIGN AND CODE IMPLEMENTATION.

"MAINTAINABLE SOFTWARE" IMPLIES THAT THE SOFTWARE PRODUCT HAS BEEN CONSTRUCTED SUCH THAT THE STRUCTURE AND ORGANIZATION OF THE SYSTEM ARE CLEAR. MODIFICATION OF THE SYSTEM CAN BE DONE WITH RELATIVE EASE (SUCH THAT CHANGES DO NOT CAUSE NEW ERRORS).

COST REDUCTION OCCURS ONLY OVER THE LIFE OF THE PRODUCT. Ada IS PRIMARILY CONCERNED WITH PROJECTS OF LONG DURATION WHICH WILL BE MODIFIED AND ENHANCED CONTINUALLY. COST SAVINGS DURING DEVELOPMENT, IF ANY, IS "ICING."

THE Ada EFFORT: DoD'S RESPONSE

THROUGH A COMBINATION OF:

- MODERN SOFTWARE ENGINEERING METHODS
- COMMON HIGH ORDER LANGUAGE (Ada)
- COMMON SUPPORT TOOLS (Ada PROGRAMMING SUPPORT ENVIRONMENT - APSE)

DEVELOP SOFTWARE THAT IS:

- RELIABLE
- MAINTAINABLE
- LESS COSTLY <u>OVER THE LIFE CYCLE</u>
- PORTABLE

INSTRUCTOR NOTES

SLIDE FORMAT IS A LIFE-CYCLE APPROACH. THE Ada LANGUAGE CAN BE VIEWED AS A SOFTWARE PRODUCT LIKE BUILDING A MISSILE: FROM ANALYSIS OF A PROBLEM AND POSSIBLE SOLUTION, THROUGH REQUIREMENTS (IN THE SERIES OF LANGUAGE REQUIREMENT SPECS), TO OPERATIONAL (WITH ACTUAL COMPILER DEVELOPMENT AND VALIDATION).

IMPORTANT TO NOTE THAT THROUGHOUT THE PROCESS, UNIVERSITIES, INDUSTRY AND COMPILER IMPLEMENTORS WERE SOLICITED FOR INPUT (REVIEWS, OPINIONS).

DEVELOPMENT OF Ada LANGUAGE

ANALYSIS	1970-1975	IDENTIFICATION OF SOFTWARE PROBLEMS IN EMBEDDED MILITARY SYSTEMS (THE CRISIS)
REQUIREMENTS	1975-1977	STRAWMAN, WOODENMAN, AND TINMAN LANGUAGE REQUIREMENTS SPECIFICATIONS
		HOLWG: HOL REQUIREMENTS FOR EMBEDDED SYSTEMS DEFINED
		EXISTING LANGUAGES EVALUATED
		RESULTS: ONE LANGUAGE IS SUFFICIENT
		NO EXISTING LANGUAGE SATISFIES ALL REQUIREMENTS
		AN EXISTING LANGUAGE SHOULD BE USED AS A BASE
DESIGN		
PHASE I	1977-1978	PRELIMINARY LANGUAGE DESIGN - IRONMAN (RED, BLUE, YELLOW, GREEN).
PHASE II	1978-1979	FORMAL LANGUAGE DEFINITION - STEELMAN (RED, GREEN)
PHASE III	1979-1980	FINAL LANGUAGE DEFINITION BY CII HONEYWELL/BULL

INSTRUCTOR NOTES

FORMAT IS A LIFE-CYCLE APPROACH. THE Ada LANGUAGE CAN BE VIEWED AS A SOFTWARE PRODUCT LIKE BUILDING A MISSILE: FROM ANALYSIS OF A PROBLEM AND POSSIBLE SOLUTION, THROUGH REQUIREMENTS (IN THE SERIES OF LANGUAGE REQUIREMENT SPECS), TO OPERATIONAL (WITH ACTUAL COMPILER DEVELOPMENT AND VALIDATION).

IMPORTANT TO NOTE THAT THROUGHOUT THE PROCESS, UNIVERSITIES, INDUSTRY AND COMPILER IMPLEMENTORS WERE SOLICITED FOR INPUT (REVIEWS, OPINIONS).

BE PREPARED TO SKIP THIS. THEY MAY ALREADY KNOW IT.

DELAUER'S PROCLAMATION SAYS THAT Ada SHALL BE USED 1 JANUARY 1984 FOR PROGRAMS ENTERING ADVANCED DEVELOPMENT AND 1 JULY 1984 FOR PROGRAMS ENTERING FULL-SCALE ENGINEERING DEVELOPMENT AS THE PROGRAMMING LANGUAGE.

LANGUAGE DEVELOPMENT (Continued)

TESTING	1980-1982	LANGUAGE REFINEMENT BY INTERNATIONAL REVIEWERS
		COMPILER VALIDATION TEST FACILITY
		ANSI STANDARDIZATION REQUESTED
OPERATIONAL	1982	COMPILER DEVELOPMENT BY DOD, PRIVATE INDUSTRY, ACADEMIA
		PARALLEL PROJECTS
	FEB. 1983	ANSI STANDARDIZATION OF Ada LANGUAGE
	MAR. 1983	NYU (Ada/ED) VALIDATED TRANSLATOR
	JUN. 1983	DR. DELAUER'S PROCLAMATION

INSTRUCTOR NOTES

WHAT DO WE MEAN BY ENVIRONMENT IN GENERAL?

ENVIRONMENTS

- PROVIDE A SET OF AUTOMATED TOOLS TO AID SOFTWARE DEVELOPERS AT VARIOUS PHASES IN THE LIFE CYCLE

  EXAMPLES: COMPILERS
  LINKERS
  LOADERS
  CODE AUDITORS
  PROGRAMMING SUPPORT LIBRARIES

- CURRENT SITUATION WITH ENVIRONMENTS

  MUST BE DEVELOPED FOR EACH MACHINE
  PERSONNEL MUST LEARN A NEW SET OF TOOLS FOR EACH MACHINE
  LIMITED TOOLS SETS AVAILABLE

SPECIFICALLY Ada ENVIRONMENTS.

THE APSE WAS INTENDED TO BE HOSTED ON ONE PHYSICAL MACHINE (GENERALLY A SIZABLE MAINFRAME WITH THE TARGET MACHINE OF THE DEVELOPMENT PROBABLY A MUCH SMALLER COMPUTER (WHICH WOULD NOT HAVE THE ADDRESS SPACE/PERIPHERALS NECESSARY)).

THE DATABASE OF THE APSE IS AN IMPORTANT FEATURE. *IT HOUSES ALL PROJECT SOURCE CODE,* OBJECT CODE, AND DOCUMENTATION.

Ada ENVIRONMENTS

- GOAL IS TO PROVIDE AUTOMATED TOOL SUPPORT FOR ALL PROJECT PERSONNEL INVOLVED IN MANAGING, DEVELOPING, AND MAINTAINING SOFTWARE SYSTEMS.
- INCLUDES TOOLS FOR ALL PHASES OF LIFE CYCLE
- ADVANTAGES

  TOOL DEVELOPMENT COSTS REDUCED

  PORTABILITY OF TOOLS, SOFTWARE, PROGRAMMERS

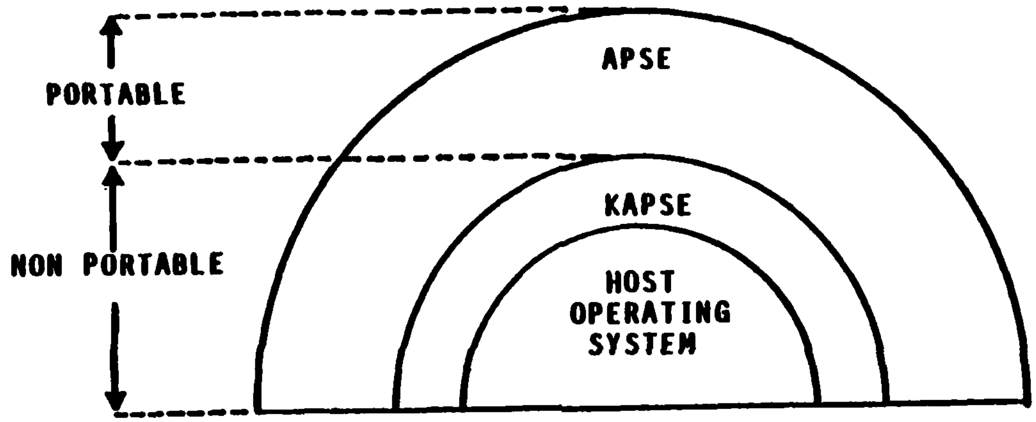
  CAN BE USED THROUGHOUT THE LIFE CYCLE

- PORTABILITY ACHIEVED THROUGH A LOW-LEVEL INTERFACE TO THE HOST OPERATING SYSTEM (THE KAPSE) AND MINIMAL SET OF TOOLS (THE MAPSE)

INSTRUCTOR NOTES

CONCEPTUALLY THE STRUCTURE IS IN NESTED LEVELS. AT THE INNERMOST LEVEL IN THE OPERATING SYSTEM IS THE PHYSICAL DATABASE. ABOVE IT, IS THE KAPSE WHICH TAKES CARE OF ALL PHYSICAL TO LOGICAL INTERFACES OF THE ENTIRE APSE. ABOVE THE KAPSE, THE MAPSE SITS. IT CONTAINS THE MINIMAL SET OF TOOLS NECESSARY TO DEVELOP SOFTWARE. SURROUNDING THE MAPSE, IS THE FULL APSE WHICH CONTAINS OTHER ADVANCED TOOLS TO BE USED TO AID DEVELOPMENT THROUGHOUT THE LIFE-CYCLE.

Ada ENVIRONMENT STRUCTURE



KAPSE:	KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT
APSE:	ADA PROGRAMMING SUPPORT ENVIRONMENT

INSTRUCTOR NOTES

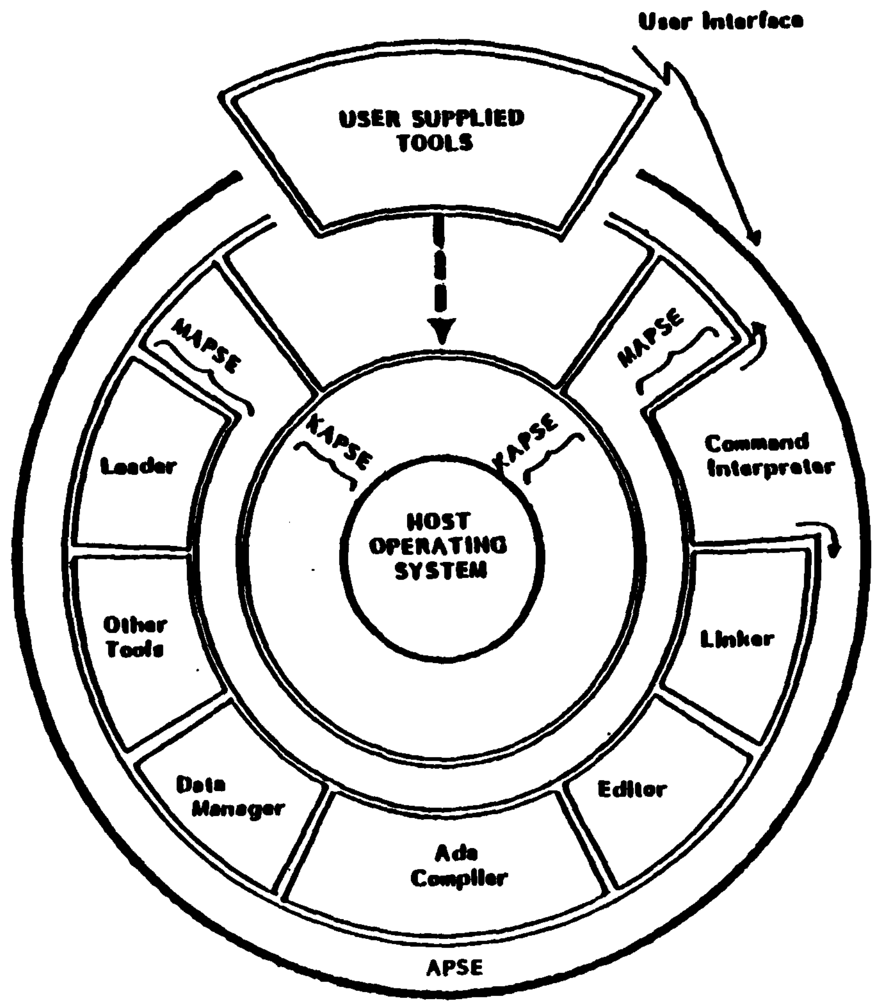
THIS IS THE COMMON PICTURE OF THE APSE STRUCTURE THAT THE STUDENT WILL SEE.

WHAT IS IN EACH PART OF APSE:

KAPSE:	NO EXPLICIT TOOLS BUT SUPPORTS - DATABASE ACCESS I/O TERMINAL TO TOOL ACCESS RUNTIME SYSTEM	
MAPSE:	COMPILER LOADER LINKER TEXT EDITOR	DYNAMIC ANALYSIS COMMAND INTERPRETER FILE ADMINISTRATOR CONFIGURATION MANAGER
APSE:	HERE SPECIFIC TOOLS HAVE NOT BEEN DETERMINED. SOME THINGS THAT MIGHT BE INCLUDED - DEBUGGERS, AUTOMATIC REQUIREMENTS/DESIGN TOOLS.	

THE KAPSE SHOULD CONTAIN ALL LOW-LEVEL FEATURES NECESSARY TO REHOST ONTO ANOTHER SYSTEM.

APSE STRUCTURE



INSTRUCTOR NOTES

SIMILAR FORMAT AS THE LANGUAGE.

OF NOTE: THE SPECIFICATION FOR THE ENVIRONMENTS IS NOT AS RIGOROUS AS FOR THE LANGUAGE SINCE WE KNOW LESS OF WHAT SHOULD BE IN AN ENVIRONMENT.

DEVELOPMENT OF Ada ENVIRONMENTS

ANALYSIS	1977-1978	LANGUAGE ALONE NOT SUFFICIENT TO IMPROVE SOFTWARE DEVELOPMENT
REQUIREMENTS	1978-1979	PRELIMINARY ENVIRONMENT REQUIREMENTS (SANDMAN, PEBBLEMAN)
DESIGN	1980	FORMAL ENVIRONMENT DEFINITION (STONEMAN)
IMPLEMENTATION	1981	COMPILER PLUS ENVIRONMENT DEVELOPMENT PROJECTS FUNDED BY DOD, PRIVATE INDUSTRY, UNIVERSITIES
TESTING	1981	KAPSE INTERFACE TEAM (KIT) / FOR INDUSTRY AND ACADEMIA (KITIA): TASK IS TO DEFINE STANDARD INTERFACES
OPERATIONAL/ MAINTENANCE	1983 →	

INSTRUCTOR NOTES

THIS RELATES THE Ada EFFORT TO OUR ORIGINAL PROBLEM. HOW OR WHY EACH PART OF THE EFFORT IS USEFUL IN ATTEMPTING TO MANAGE OUR SOFTWARE PROBLEMS. IN THIS PERSPECTIVE, Ada IS NOT JUST A LANGUAGE, BUT BECOMES A TOOL - LIKE LINKERS, DEBUGGERS, METHODOLOGIES - TO DEAL WITH SOFTWARE DEVELOPMENT PROBLEMS.

RELIABILITY AND MAINTAINABILITY ARE INCREASED THROUGH MODERN SOFTWARE ENGINEERING PRINCIPLES AND METHODS SUCH AS STRUCTURED DESIGN AND PROGRAMMING (WHICH ALSO HELP INCREASE PRODUCTIVITY), MODULARITY, STRONG TYPING AND ERROR RECOVERY MECHANISMS.

THE Ada EFFORT AND THE SOFTWARE CRISIS

- MODERN SOFTWARE ENGINEERING METHODS
  - INCREASED PRODUCTIVITY
  - INCREASED RELIABILITY, MAINTAINABILITY
- COMMON HIGH ORDER LANGUAGE
  - DESIGNED TO SUPPORT MODERN SOFTWARE DEVELOPMENT METHODS
  - SUPPORTS THE MANAGEMENT OF COMPLEXITY AND CHANGING REQUIREMENTS
  - REDUCED PROGRAMMER RETRAINING
- COMMON SUPPORT ENVIRONMENT
  - REDUCED COST OF WRITING CUSTOMIZED SYSTEMS PROGRAMS
  - INCREASED PORTABILITY OF SOFTWARE/PROGRAMMERS
  - LIFE CYCLE SUPPORT OF SOFTWARE DEVELOPMENT
  - REDUCED PROGRAMMER RETRAINING

INSTRUCTOR NOTES

THIS SECTION PROVIDES AN OVERVIEW (CONCEPTUAL, INTUITIVE FEEL) OF PROGRAMMING IN Ada FROM PROBLEM DEFINITION TO MAINTENANCE.

STRESS TO THE STUDENTS THAT SYNTAX IS NOT THE KEY ISSUE HERE - OVERALL STRUCTURE AND CONCEPTS IS.

TOPIC OUTLINE

BACKGROUND AND RATIONALE FOR Ada

WRITING AN Ada PROGRAM FROM BEGINNING TO END

SUMMARY OF Ada FEATURES

INSTRUCTOR NOTE

POINT OUT THE FOURTH STEP. IT IS AS IMPORTANT (IF NOT MORE SO) THAN THE OTHER THREE.

OUR PROCESS

STATEMENT OF REQUIREMENTS

DECOMPOSITION OF SOLUTION

Ada IMPLEMENTATION (CODE AND COMPILATION)

CHANGES TO THE SYSTEM

INSTRUCTOR NOTE

THE PURPOSE OF THE EXAMPLE IS TO ILLUSTRATE WHAT IT'S LIKE TO WRITE AN Ada PROGRAM FROM BEGINNING TO END. THIS EXAMPLE IS ELEMENTARY BUT BECAUSE OF THAT, THE STUDENT CAN CONCENTRATE ON THE Ada AND NOT THE ALGORITHMS. THE FORMAT IS TO PARALLEL THE SOFTWARE DEVELOPMENT CYCLE. FIRST DECOMPOSE THE PROBLEM FROM THE TOP, DOWN THROUGH SPECIFIC ALGORITHMS TO THE CONTROL STRUCTURE LEVEL. AFTER ANALYZING THE PROBLEM, THE Ada CODE IS BUILT FROM THIS POINT BACK UP TO A COMPLETE Ada SYSTEM. THE Ada SYNTAX IS TOTALLY BY EXAMPLE (I.E. OSMOSIS). ADDITIONAL GOALS ARE TO GENERATE A FAMILIARITY WITH Ada, THE EASE WITH WHICH IT CAN BE READ, AND TO CREATE A NON-THREATENING APPRECIATION FOR THE LANGUAGE. TO BUILD AN Ada SYSTEM, WE START FIRST WITH CONTROL STRUCTURES, AS ACTION STATEMENTS IN Ada ARE VERY SIMILAR TO OTHER LANGUAGES. THE CODE FRAGMENTS ARE SIMILAR TO WHAT WILL BE USED IN THE FINAL CODE. IN THIS WAY THE RATIONALE IS SET FOR TYPES AND OBJECTS. NEXT, A LOOK AT TYPE AND OBJECT DECLARATIONS. AGAIN ACTUAL CODE RELATED TO THE EXAMPLE IS USED. CODE COMMENTS PROVIDE EXPLANATIONS OF THE Ada THUS AFTER THE COURSE IS FINISHED THE STUDENT CAN REFER BACK TO THE COURSE NOTES WITH UNDERSTANDING. THE EXAMPLE NOW BUILDS TO Ada SUBPROGRAMS AND PARAMETERS. AT THIS POINT, THE COMPLETED CODE IS PRESENTED FOR ALL PROCEDURES AND FUNCTIONS. NEXT THESE RESOURCES ARE COLLECTED INTO AN Ada PACKAGE. Ada PROVIDES THE FACILITIES TO CREATE OUR OWN PACKAGES. AT THIS POINT THE STUDENTS SHOULD HAVE AN INTUITIVE FEEL FOR THE USEFULNESS OF THE PACKAGE CONCEPT IN Ada. FINALLY, THE LOGIC OF THE MAIN PROCEDURE IS PRESENTED, WHICH USES THE RESOURCES OF TWO PACKAGES. WITHIN THE MAIN PROCEDURE, A SIMPLE I/O FORMAT IS PRESENTED TO ILLUSTRATE BOTH THE ABILITY TO CREATE ONE'S OWN I/O ROUTINES, SPECIALLY TAILORED, AND TO ALSO SHOW THE USE OF THE 'GET' AND 'PUT' PROCEDURES. AS A WHOLE THE Ada EXAMPLE ILLUSTRATES A BASIC PROGRAM STRUCTURE - I.E. A MAIN DRIVER PROCEDURE USING RESOURCES FROM ONE OR MORE PACKAGES WITH THE PACKAGES IN TURN CONSISTING OF NESTED SUBPROGRAMS. AS PART OF CODING Ada, THE SYSTEM MUST BE COMPILED TO TRANSLATE THE SOURCE CODE INTO OBJECT CODE FOR EVENTUAL EXECUTION. COMPILATION AND THE PROGRAM LIBRARY ARE PRESENTED FOLLOWED BY TWO EXAMPLES OF CHANGES TO THE SYSTEM.

STATEMENT OF THE PROBLEM

A SYSTEM THAT RECORDS AND TRACKS TWO-DIMENSIONAL MOVEMENT ON A RADAR SCREEN NEEDS A PROCEDURE THAT, GIVEN THE LAST POSITION RECORDED, THE CURRENT POSITION, THE TIME BETWEEN THOSE READINGS, AND A NEW TIME INTERVAL, WILL PREDICT WHERE THE NEXT POINT SHOULD OCCUR. THE PREDICTION WILL ASSUME THAT NO CHANGE IN SPEED OR DIRECTION WILL OCCUR; THE VALUE THUS OBTAINED MIGHT LATER BE COMPARED TO THE ACTUAL READING TO DETERMINE PATTERNS OF CHANGE IN EITHER FACTOR. THE TRACKING PROGRAM THUS NEEDS ACCESS TO A NEXT-POINT CALCULATION ROUTINE, WHICH SHOULD BE ASSOCIATED WITH FACILITIES TO CALCULATE THE DISTANCE BETWEEN TWO POINTS AND TO DETERMINE VELOCITY. DUE TO THE SPECIFICS OF THE SYSTEM, A VENDOR-SUPPLIED PACKAGE CONTAINING SUCH ROUTINES WOULD BE UNSUITABLE.

INSTRUCTOR NOTES

FOR THE EXAMPLE WE ARE NOT TRYING TO SHOW THE BEST OR ONLY WAY TO APPROACH THE PROBLEM BUT RATHER TO ILLUSTRATE THE THOUGHT PROCESS INVOLVED IN Ada SYSTEMS.

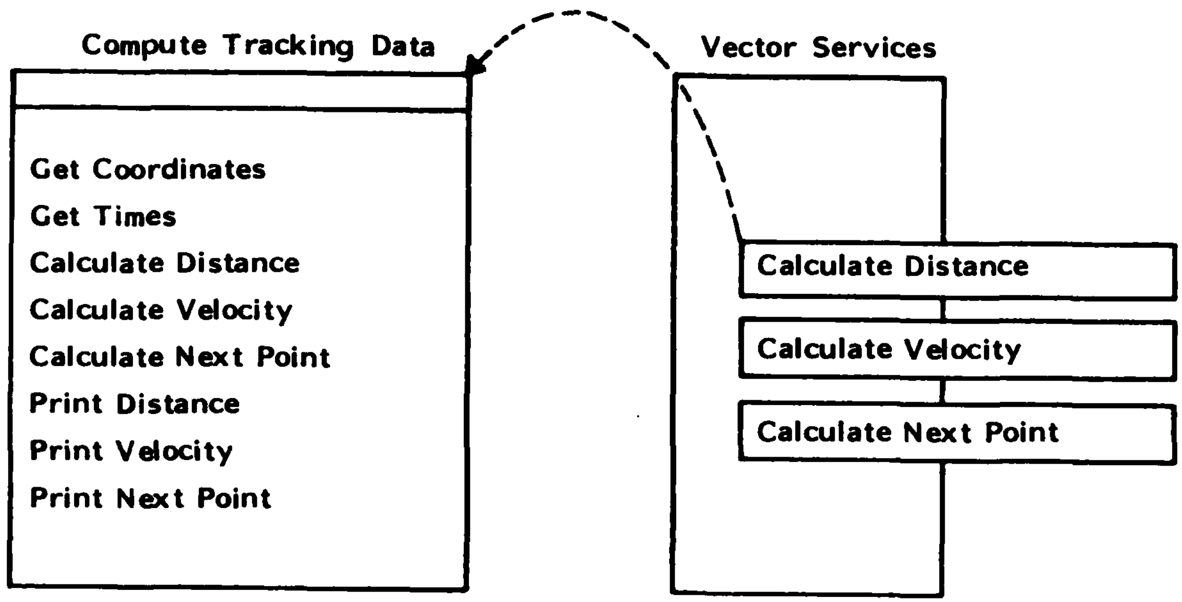
WE BEGIN AT A HIGH LEVEL OF ABSTRACTION OF THE PROBLEM AND CONTINUE TO DECOMPOSE TO THE STATEMENT LEVEL.

"LET US SUMMARIZE THE OBJECTS TO BE DEALT WITH AND THE OPERATIONS NEEDED TO BE PERFORMED RELATIVE TO THE OBJECTS."

A PICTURE OF A SOLUTION IS SHOWN. IT HAS BEEN DECIDED TO HAVE A MAIN PROGRAM WHICH CONTROLS THE OVERALL LOGIC FLOW OF THE SYSTEM. A SMALL PACKAGE WILL IMPLEMENT THE VECTOR CALCULATIONS. THE MAIN PROCEDURE LOGIC IS PRESENTED AS PSEUDO-CODE FOR THE MOMENT. BUT THE POSSIBLE SOLUTION MUST BE FURTHER DECOMPOSED TO UNDERSTAND THE VECTOR SERVICES MORE FULLY.

DECOMPOSITION OF SOLUTION: TRACKING PROGRAM

OBJECTS	OPERATIONS
POINTS	CALCULATE DISTANCE
TIMES	CALCULATE VELOCITY
	CALCULATE NEXT POINT



INSTRUCTOR NOTES

A LOOK AT THE DISTANCE FUNCTION IS FIRST. A FORMULA FOR CALCULATING THE DISTANCE BETWEEN TWO POINTS IS GIVEN. AGAIN LOOKING AT THE OBJECTS, LAST POINT AND THIS POINT, DETERMINE THE OPERATIONS NECESSARY FOR THE CALCULATION. ONCE AGAIN WE DECOMPOSE THE SOLUTION INTO A CONTROL LOGIC AND A PACKAGE OF GENERAL SERVICES. SIMILAR DECOMPOSITION WOULD OCCUR FOR ALL OTHER SUBROUTINES.

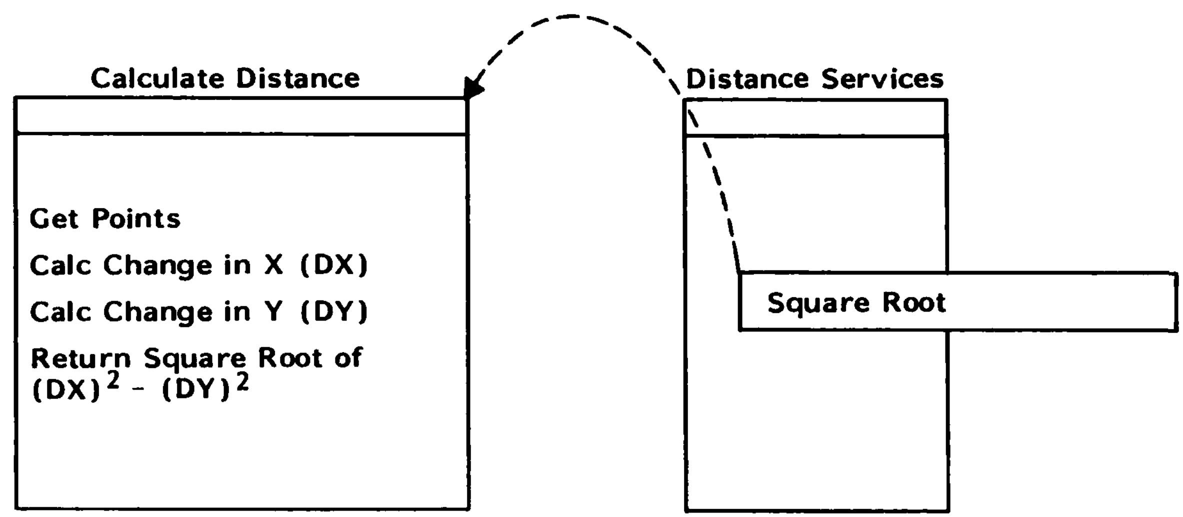
POINT OUT THAT Ada HAS NO *PREDEFINED* SQUARE ROOT FUNCTON, SO ONE MUST BE PROVIDED HERE. IT MIGHT ALTERNATIVELY BE INCLUDED IN A MATH SERVICES PACKAGE.

SINCE CALCULATING CHANGE IN X AND Y VALUES INVOLVES ONLY SIMPLE SUBTRACTION, NO ADDITIONAL SUBROUTINES ARE ADDED TO THE DESIGN; INLINE CODE IS MORE EFFICIENT.

CALCULATE DISTANCE

$$\text{DISTANCE} = \sqrt{(\text{CHANGE IN X})^2 - (\text{CHANGE IN Y})^2}$$

OBJECTS	OPERATIONS
LAST POINT	CHANGE IN X AND Y
THIS POINT	SQUARE ROOT

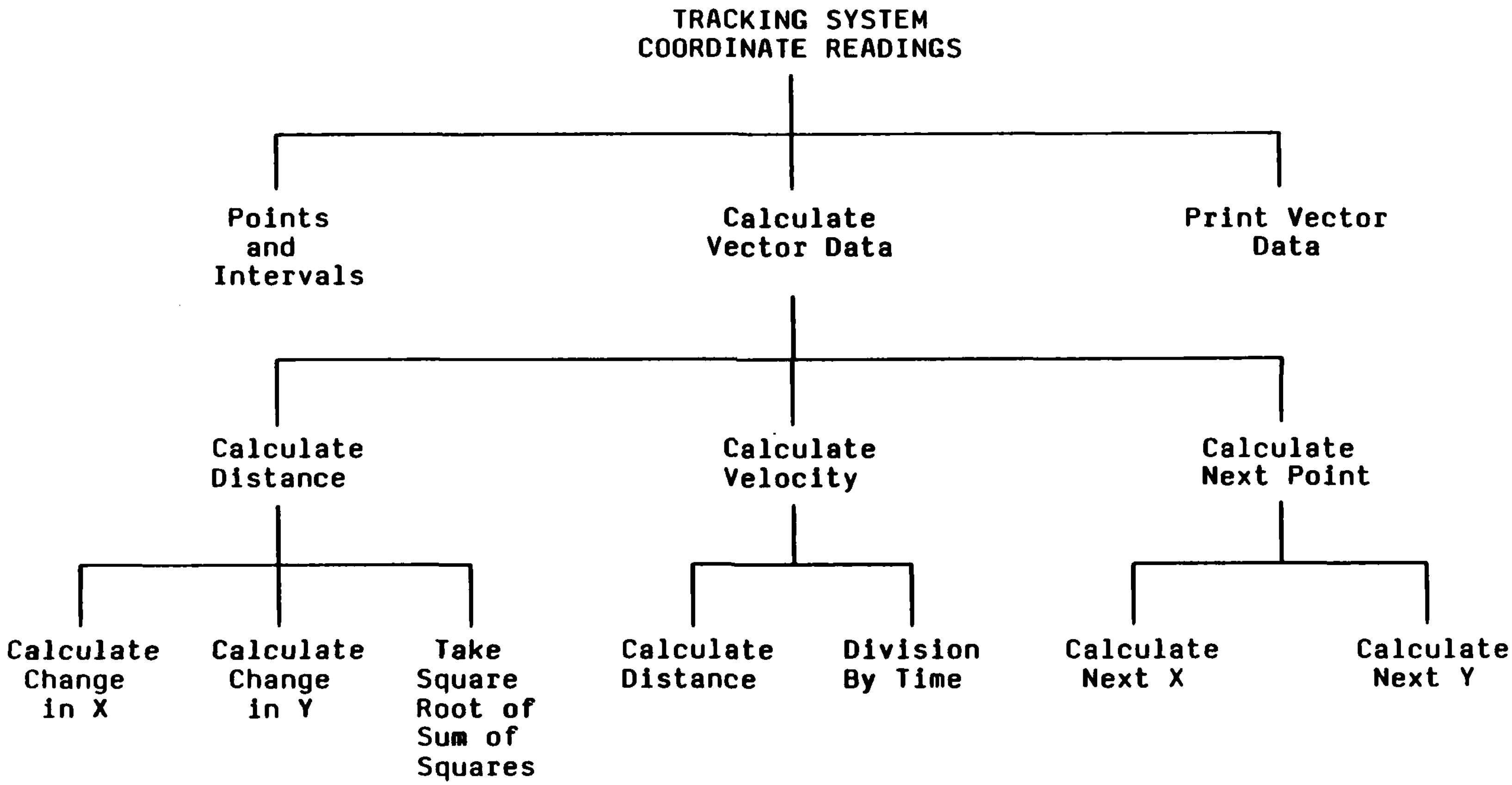


INSTRUCTOR NOTES

THE DIAGRAM SUMMARIZES THE LEVELS OF DECOMPOSITION OF THE SAMPLE DESIGN.

WE NOW TURN TO THE ACTUAL Ada CODING PHASE.

DESIGN SOLUTION SUMMARY



INSTRUCTOR NOTES

THE LISTED Ada FEATURES WILL BE DISCUSSED AS PREPARATION IS MADE FOR THE CODING OF THE SOLUTION.

Ada FEATURES FOR SOLUTION

AS WE EXPRESS OUR SOLUTION FOR A TRACKING PROGRAM IN Ada, WE MUST LOOK AT:

- PACKAGES
- SUBPROGRAMS
- TYPES AND DECLARATIONS
- CONTROL STRUCTURES AND STATEMENTS

INSTRUCTOR NOTES

PACKAGES HAVE TWO PARTS. THE FIRST IS CALLED THE SPECIFICATION. IT TELLS WHAT KINDS OF ACTIONS OR DATA CAN BE USED.

FOR THE EXAMPLE, THE FORM OF THE DATA (TYPE) PLUS THE LIST OF DECLARATIONS OF THE SUBPROGRAMS PROVIDING RESOURCES FORM THE SPECIFICATION.

THIS PACKAGE, CALLED Vector_Services, CONTAINS THE TRACKING RESOURCES NEEDED BY OUR PROGRAM. SINCE THE SPECIFICATION SHOWN PROVIDES ALL INFORMATION NECESSARY TO USE THE RESOURCES, WE CAN NOW TURN TO THE MAIN PROGRAM, EVEN THOUGH WE DON'T YET KNOW HOW THESE RESOURCES ARE ACTUALLY IMPLEMENTED.

PACKAGES

SPECIFICATION

```
package Vector_Services is

   type Coordinate_Type is (X, Y);

   type Point_Type is array (Coordinate_Type) of Float;

   subtype Time_Type is Duration;

   function Distance_Between (Last_Point, This_Point : Point_Type)
                     return Float;

   procedure Calculate_Velocity (From, To : in Point_Type;
                                 In_Time  : in Time_Type;
                                 Velocity : out Float);

   function Next_Point_After
               (Last_Point, This_Point : in Point_Type;
                Time_Between_Last, Time_Between_Next : Time_Type)
                return Point_Type;

end Vector_Services;
```

INSTRUCTOR NOTES

IN THE SECOND PART OF THE PROGRAM UNIT, THE BODY, IS THE ACTUAL CODE THAT PERFORMS THE ACTIONS OF THE RESOURCES.

NOTICE THAT PROCEDURE Sqrt WAS NOT LISTED IN THE SPECIFICATION. Sqrt IS A UTILITY WHICH WILL ONLY BE USED BY THE ALGORITHM Distance_Between. BY PLACING IT IN THE PACKAGE BODY, IT ENSURES THAT NO UNAUTHORIZED TAMPERING OF THE DATA CAN BE DONE.

WHERE THE STUDENT SEES THE ... IS WHERE THE SUBPROGRAM CODE WOULD BE PLACED.

PACKAGES (Continued)

BODY

```
package body Vector_Services is

   function Sqrt (X : Float) return Float is ...;

   function Distance_Between (Last_Point, This_Point : Point_Type)
              return Float is ...;

   procedure Calculate_Velocity (From, To : in  Point_Type;
                                 In_Time  : in  Time_Type;
                                 Velocity : out Float) is ...;

   function Next_Point_After (Last_Point, This_Point : in Point_Type;
                              Time_Between_Last, Time_Between_Next : Time_Type);
                    return Point_Type is ...;

end Vector_Services;
```

INSTRUCTOR NOTES

THE Ada SYSTEM CAN NOW BE FURTHER DEVELOPED BY CODING THE MAIN LOGIC PROCEDURE. THE TRACKING RESOURCES ARE PROVIDED IN THE Vector_Services PACKAGE SHOWN ON THE PREVIOUS SLIDE. SO THE MAIN PROCEDURE KNOWS ABOUT THESE RESOURCES, THE with CLAUSE MUST BE USED. THE RESOURCES FROM AN I/O PACKAGE CALLED Text_IO WILL ALSO BE USED.

PROCEDURE Compute_Tracking_Data HAS THE SAME FORMAT AS THE OTHER PROCEDURES (EXCEPT IT HAS NO PARAMETERS). THIS SLIDE SHOWS THE DECLARATIONS FOR ALL DATA OBJECTS AND LOCAL ROUTINES TO BE USED IN THE STATEMENT PART. THE USE OF "is separate" WILL BE DISCUSSED IN LATER SLIDES.

(IF POSSIBLE, DISPLAY THIS SLIDE AND THE NEXT AT THE SAME TIME).

MAIN PROGRAM LOGIC

```
with Text_IO, Vector_Services;
use Vector_Services;
procedure Compute_Tracking_Data is

   Last_Point, Current_Point, Next_Point : Point_Type;
   Time_Elapsed, Time_Projected : Time_Type;
   Distance, Velocity : Float;

   package Time_IO is new Text_IO.Fixed_IO (Time_Type);
   package Flt_IO is new Text_IO.Float_IO (Float);

   procedure Get_Point (P : out Point_Type) is separate;
   procedure Put_Point (P : in  Point_Type) is separate;

begin -- Compute_Tracking_Data

                  EXECUTABLE PART ON NEXT PAGE

end Compute_Tracking_Data;
```

INSTRUCTOR NOTES

THIS SLIDE SHOWS THE STATEMENT PART OF Compute_Tracking_Data: STATEMENTS TO READ THE POINTS AND TIMES WITH THE SERVICES OF Text_IO; THE DESIRED INFORMATION IS CALCULATED VIA FUNCTION AND PROCEDURE CALLS; AND WE PRINT OUR RESULTS USING THE SERVICES OF Text_IO. NOTE THE SUBSTITUTION OF ACTUAL PARAMETERS FOR THE FORMAL PARAMETERS OF THE SUBPROGRAM DEFINITIONS.

IF ASKED, Calculate_Velocity IS A PROCEDURE RATHER THAN A *FUNCTION* FOR THE PURPOSE OF ILLUSTRATING THE FORM OF A PROCEDURE DECLARATION IN CONTRAST TO THAT OF A FUNCTION. IT WOULD PROBABLY BE BETTER CODED AS A FUNCTION.

MAIN PROGRAM LOGIC (Continued)

```
with Text_IO, Vector_Services;
use Vector_Services;
procedure Compute_Tracking_Data is
```

DECLARATIVE PART ON PREVIOUS PAGE

```
begin  -- Compute_Tracking_Data
   Text_IO.Put ("Enter coordinates of last position: ");
   Get_Point (Last_Point);
   Text_IO.Put ("Enter coordinates of current position: ");
   Get_Point (Current_Point);

   Text_IO.Put ("Time (in seconds) between readings : ");
   Time_IO.Get (Time_Elapsed);     Text_IO.New_Line;
   Text_IO.Put ("Time (in seconds) until next reading : ");
   Time_IO.Get (Time_Projected);   Text_IO.New_Line;

   Distance := Distance_Between (Last_Point, Current_Point);
   Calculate_Velocity (Last_Point, Current_Point, Time_Elapsed, Velocity);
   Next_Point := Next_Point_After (Last_Point, Current_Point,
                                   Time_Elapsed, Time_Projected);

   Text_IO.Put ("Distance between points was ");
   Flt_IO.Put (Distance);
   Text_IO.Put_Line (" units.");

   Text_IO.Put ("Velocity was ");
   Flt_IO.Put (Velocity);
   Text_IO.Put_Line (" units per second.");

   Text_IO.Put ("After ");
   Time_IO.Put (Time_Projected);
   Text_IO.Put ("seconds, the next point should be ");
   Put_Point (Next_Point);

end Compute_Tracking_Data;
```

INSTRUCTOR NOTES

CODING OF THE Ada SYSTEM IS COMPLETED. NEXT THE TOPIC OF COMPILATION IN Ada IS DISCUSSED.

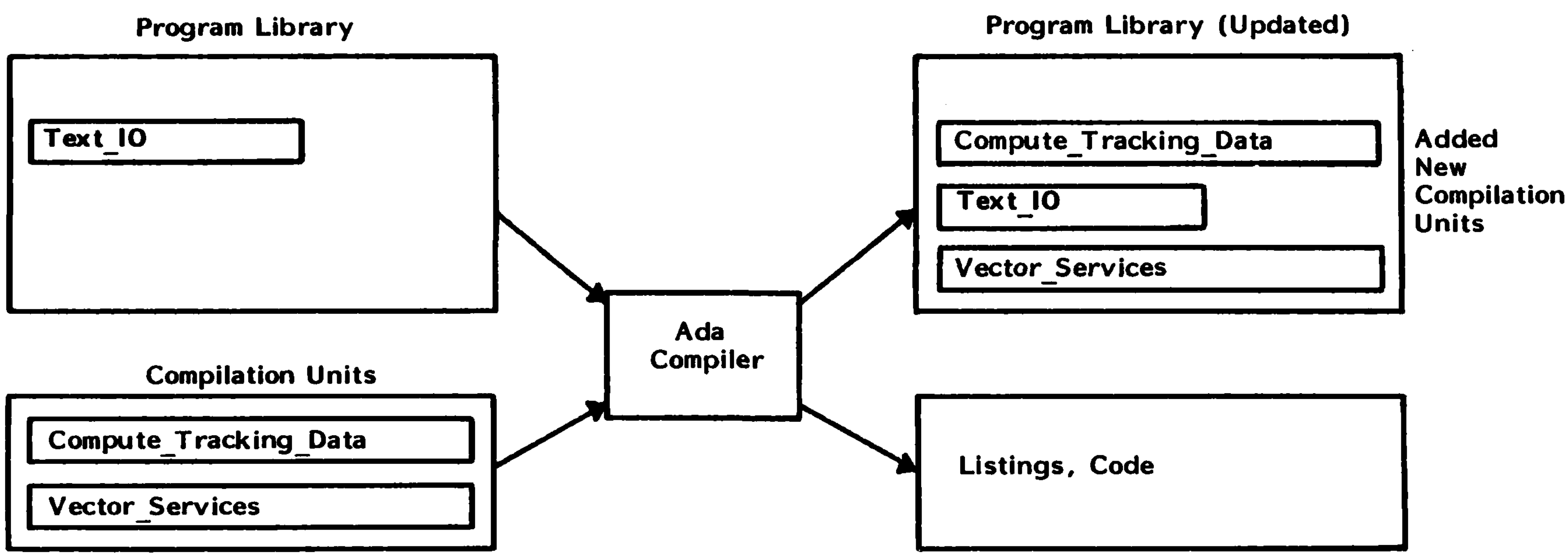
COMPILATION UNITS ARE PARTS OF Ada CODE THAT THE LANGUAGE SAYS CAN BE SUBMITTED BY THEMSELVES TO AN Ada COMPILER.

COMPILATION CONSISTS OF SUBMITTING OUR COMPILATION UNITS INTO A PROGRAM LIBRARY WHICH IS A FILE THAT WILL CONTAIN CERTAIN INFORMATION THAT SUBSEQUENT COMPILER SUBMISSIONS WILL NEED. ONCE COMPILED, THE SUBMITTED COMPILATION UNITS ARE ADDED TO THE PROGRAM LIBRARY.

CODE COULD BE INTERMEDIATE SOURCE CODE OR OBJECT CODE.

COMPILATION OF TRACKING SYSTEM

- SUBMIT ALL PROGRAM PARTS AT ONE TIME:

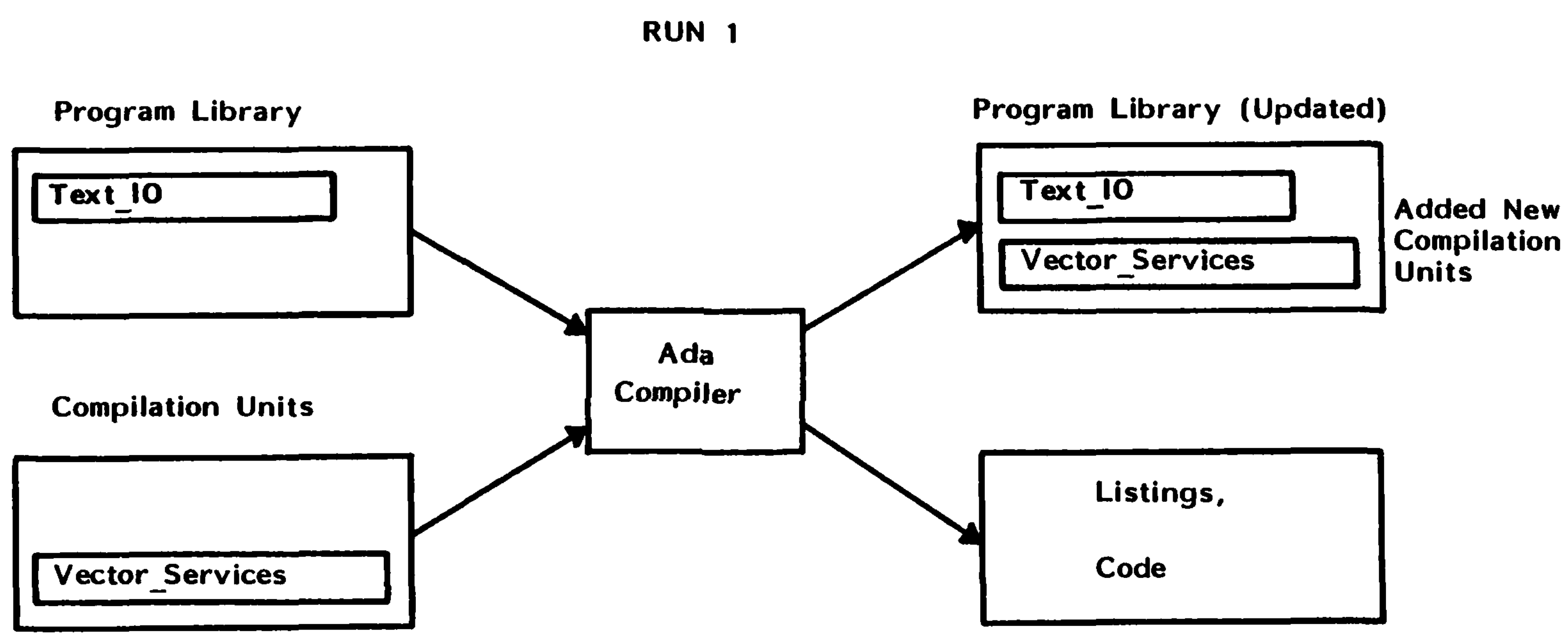


INSTRUCTOR NOTES

INSTEAD OF SUBMITTING ALL OUR PROGRAM PARTS AT ONE TIME, WE COULD SUBMIT THEM SEPARATELY. LET'S SAY PROGRAMMER 1 CODED OUR Vector_Services PACKAGE. INSTEAD OF WAITING FOR PROGRAMMER 2, WHO WILL HAVE HIS CODE COMPLETED LATER, WE CAN COMPILE THE Vector_Services PACKAGE INTO THE PROGRAM LIBRARY.

ALTERNATE COMPILATION OF TRACKING SYSTEM

- SUBMIT PROGRAM PARTS (COMPILATION UNITS) SEPARATELY:

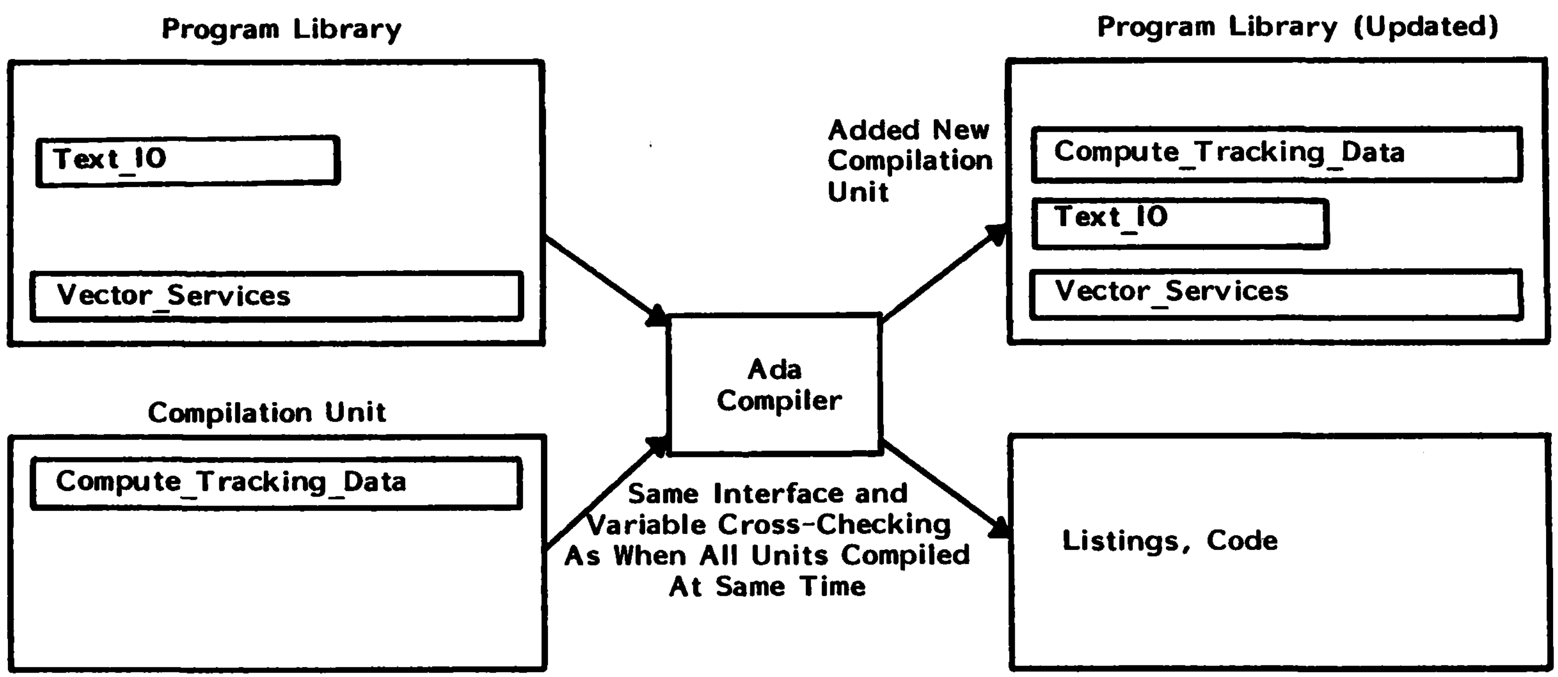


INSTRUCTOR NOTES

WHEN PROGRAMMER 2 IS FINISHED, WE THEN COMPILE OUR PROCEDURE Compute_Tracking_Data INTO THE PROGRAM LIBRARY. WITH THE INFORMATION CONTAINED IN THE PROGRAM LIBRARY, THE COMPILER CAN DO THE SAME INTERFACE AND VARIABLE CROSS-CHECKING BETWEEN Compute_Tracking_Data AND THE PACKAGE - JUST AS IF THEY HAD BEEN COMPILED AT THE SAME TIME.

ALTERNATE COMPILATION (Continued)

RUN 2



THIS WAY IS CALLED SEPARATE COMPILATION.

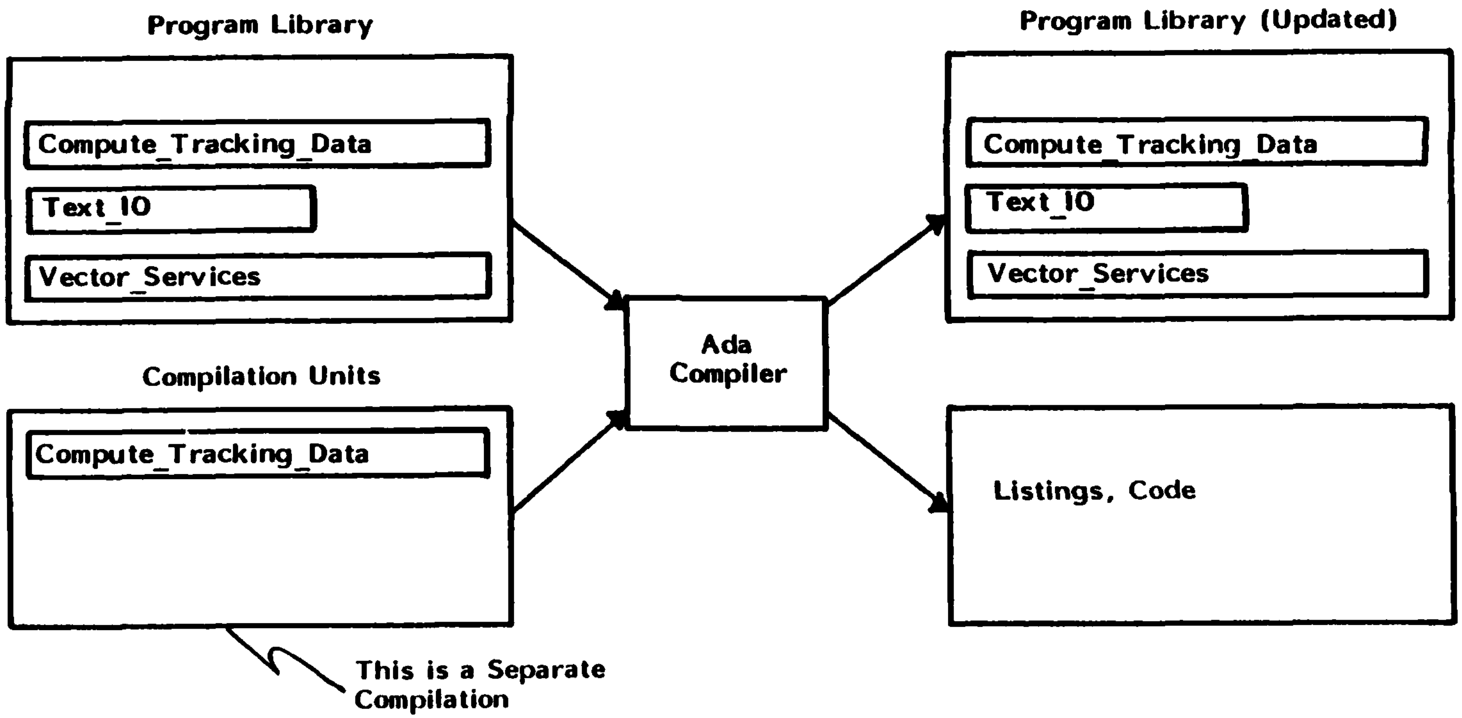
INSTRUCTOR NOTES

THE NATURE OF LARGE SYSTEMS IS CONTINUAL CHANGE. WE NEXT LOOK A HOW THAT CAN AFFECT OUR SOLUTION.

THE GOAL OF THIS SLIDE IS TO ILLUSTRATE ONE OF THE GREAT ADVANTAGES OF Ada - THE PACKAGE - FOR LOCALIZATION OF EFFECT OF CHANGES.

CHANGES TO THE SYSTEM: MAIN PROCEDURE

WE WANT TO CHANGE THE I/O FORMAT IN Compute_Tracking_Data. WE MAKE THE CHANGE. BECAUSE WE PACKAGED OUR COMPUTATIONAL RESOURCES, AND WE DON'T HAVE TO CHANGE THOSE RESOURCES, WE DON'T HAVE TO CHANGE OR RECOMPILE THE VECTOR PACKAGE.



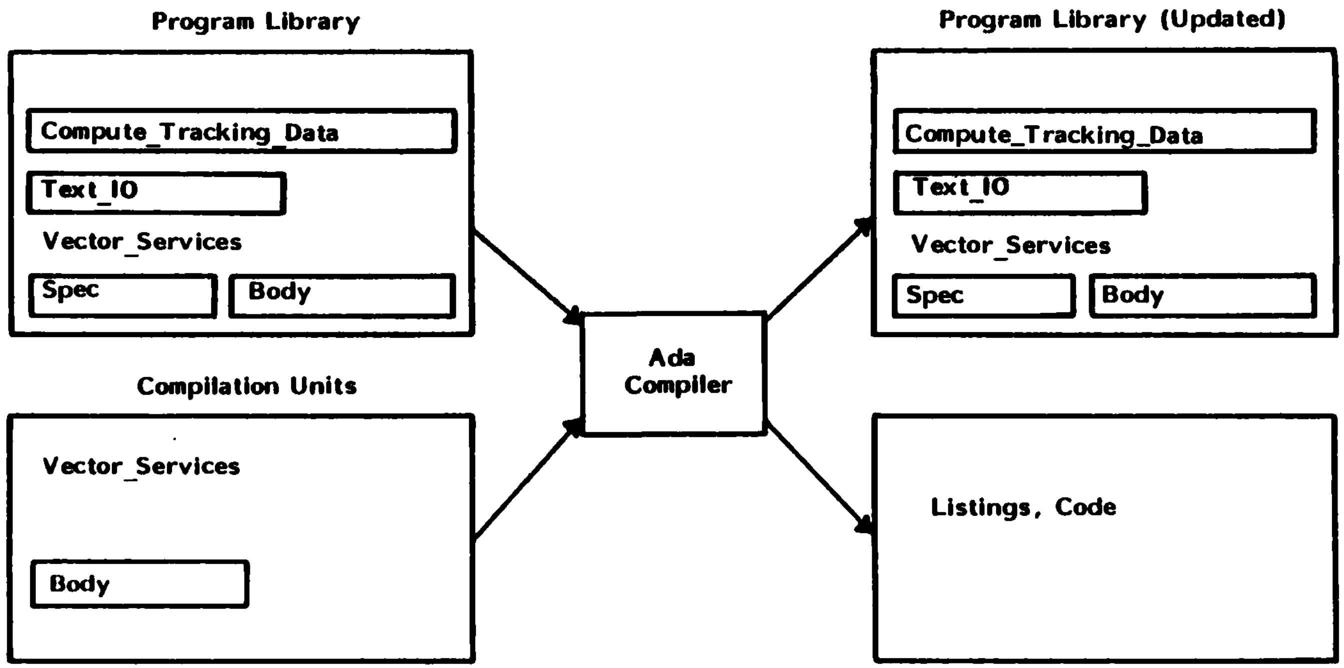
INSTRUCTOR NOTES

THE GOAL OF THIS SLIDE IS TO ILLUSTRATE FURTHER ADVANTAGES OF Ada FEATURES - THE PACKAGE FOR PORTABILITY OR REUSABILITY.

POINT OUT THAT NEITHER Compute_Tracking_Data NOR THE PACKAGE SPECIFICATION FOR Vector_Services NEED TO BE RECOMPILED.

CHANGES TO THE SYSTEM: PACKAGE BODY

WE FIND A BETTER ALGORITHM FOR ONE OF OUR VECTOR ROUTINES. SINCE WE COLLECTED OUR VECTOR ROUTINES IN A PACKAGE, WE CAN MAKE THE CHANGE TO THE PACKAGE BODY Vector_Services WITHOUT REQUIRING ANY CHANGES TO THE MAIN PROCEDURE OR THE PACKAGE SPECIFICATION FOR Vector_Services.

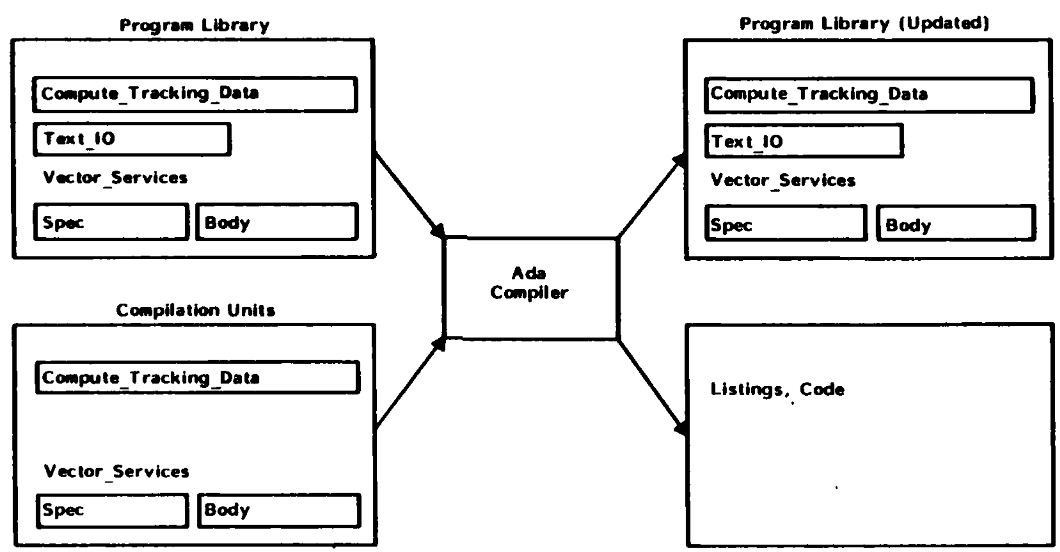


INSTRUCTOR NOTES

CHANGES TO THE SYSTEM: ADDING A ROUTINE

WE WANT TO ADD A ROUTINE TO COMPUTE THE ANGLE OF THE VECTOR. SINCE WE COLLECTED OUR VECTOR ROUTINES IN A PACKAGE, WE WANT TO ADD THIS ROUTINE TO THE PACKAGE SPECIFICATION AND BODY OF Vector_Services. WE MODIFY Vector_Services, AND Compute_Tracking_Data DEPENDS ON THOSE RESOURCES.

AS A RESULT WE MUST ALSO RECOMPILE Compute_Tracking_Data.



INSTRUCTOR NOTES

THE NEXT SEVERAL SLIDES SET UP THE MOTIVATION FOR AND ILLUSTRATE THE USE OF SUBUNITS.

WE'D LIKE TO KEEP OUR SYSTEM UNDERSTANDABLE AND READABLE. IN PURSUING THIS GOAL, WE RETURN TO THE PACKAGE BODY OF Vector_Services.

INSTRUCTOR NOTES

DON'T GO THROUGH THE CODE IN DETAIL. THIS IS THE PACKAGE BODY AS IT WOULD 'REALLY' APPEAR. NOTE HOW CONFUSING IT IS - WE CAN'T AS EASILY SEE WHAT FUNCTIONS, SUBPROGRAMS, ETC. ARE CONTAINED IN THE BODY.

POINT OUT THE CALL TO Sqrt IN Distance_Between (I.E. Distance_Between DEPENDS ON Sqrt, WHICH AFFECTS COMPILATION ORDER).

THE COMPLETE PACKAGE BODY

```
package body Vector_Services is

   function Sqrt (X : Float) return Float is
      Epsilon : constant := 0.000001;
      Root : Float := 1.0;
   begin -- Sqrt
      if X = 0.0 then
         return 0.0;
      else
         Root := (X/Root + Root) / 2.0;
         while abs (X/Root**2 - 1.0) > = Epsilon
            loop
               Root := (X/Root + Root) / 2.0;
            end loop;
         return Root;
      end if;
   end Sqrt;

   function Distance_Between (Last_Point, This_Point : Point_Type) return Float is
      Dx, Dy : Float;
   begin -- Distance_Between
      Dx := abs (This_Point(X) - Last_Point(X));
      Dy := abs (This_Point(Y) - Last_Point(Y));
   end Distance_Between;
```

INSTRUCTOR NOTES

POINT OUT THE CALL TO Distance_Between IN THE BODY OF Calculate_Velocity. AGAIN, THIS WILL HAVE AN EFFECT ON COMPILATION DECISIONS.

PACKAGE BODY (Continued)

```
procedure Calculate_Velocity (From, To : in Point_Type;
                              In_Time  : in Time_Type;
                              Velocity : out Float) is
begin -- Calculate_Velocity
     Velocity := Distance_Between(From, To)/ Float(In_Time);
end Calculate_Velocity;

function Next_Point_After (Last_Point, This_Point : in Point_Type;
                           Time_Between_Last, Time_Between_Next : Time_Type)
          return Point_Type is

     Next_Point : Point_Type;

begin -- Next_Point_After

   if Time_Between_Last = 0 then
      return This_Point;
   else
      Next_Point(X) := Last_Point (X) + Float(Time_Between_Next/Time_Between_Last)
                          * abs (This_Point(X) - Last_Point(X));
      Next_Point(Y) := Last_Point (Y) + Float(Time_Between_Next/Time_Between_Last)
                          * abs (This_Point(Y) - Last_Point(Y));
      return Next_Point;
   end if;

end Next_Point_After;

end Vector_Services;
```

INSTRUCTOR NOTES

Ada ALLOWS US TO CAPTURE THE INITIAL STRUCTURE AND COMPOSITION OF THE PACKAGE BODY THROUGH STUBBING.

'is separate' JUST SAYS TO THE COMPILER, "YOU WILL FIND THE ACTUAL CODE FOR THIS SUBPROGRAM IN A SEPARATE PLACE FROM THE PARENT (OR CONTAINING) Ada UNIT".

IN CONCEPT, STUBBING IS SIMILAR TO SUBROUTINES IN FORTRAN, ASSEMBLY LANGUAGE, SERIAL.

RECALL THAT STUBS WERE ALSO FOUND IN Compute_Tracking_Data IN DECLARING Get_Point AND Put_Point.

- TO CAPTURE THE 'STRUCTURE' OF THE PACKAGE BODY

```
package body Vector_Services is
   function Sqrt (X : Float) return Float is separate; -- A STUB
   function Distance_Between (Last_Point, This_Point : Point_Type) return Float
                             is separate;
   procedure Calculate_Velocity (From, To : in Point_Type;
                                 In_Time  : in Time_Type;
                                 Velocity : out Float) is separate;
   function Next_Point_After (Last_Point, This_Point : in Point_Type;
                             Time_Between_Last, Time_Between_Next : Time_Type)
                             return Point_Type is separate;

end Vector_Services;
```

INSTRUCTOR NOTES

SUBUNITS

- IN ADDITION, FOR EACH 'SEPARATE' SUBPROGRAM (SUBUNIT) WE INDICATE THE PARENT UNIT AS FOLLOWS:

```
separate (Vector_Services)    -- We Add This Line
function Sqrt (X : Float) return Float is
   Epsilon : constant := 0.000001;
   Root : Float := 1.0;
begin -- Sqrt
   if X = 0.0 then
      return 0.0;
   else
      Root := (X/Root + Root) / 2.0;
      while abs (X/Root**2 - 1.0) > = Epsilon
         loop
            Root := (X/Root + Root) / 2.0;
         end loop;
      return Root;
   end if;
end Sqrt;
```

INSTRUCTOR NOTES

THESE ARE THE SUBUNITS STUBBED OUT OF THE MAIN PROCEDURE. NOTE THAT THIS CODE WOULD ADD CONSIDERABLE BULK TO THE MAIN PROCEDURE BODY IF USED INLINE, WHILE CONTRIBUTING LITTLE TO THE LOGICAL STRUCTURE. STUBBING OUT THESE ROUTINES ALLOWS EASY MODIFICATION OF I/O FORMAT.

MORE SUBUNITS

```
separate (Compute_Tracking_Data)
procedure Get_Point (P : out Point_Type) is
begin -- Get_Point
   Text_IO.Put ("  X = ");
   Flt_IO.Get (P(X));
   Text_IO.Put (";  Y = ");
   Flt_IO.Get (P(Y));
   Text_IO.New_Line;
end Get_Point;

separate (Compute_Tracking_Data)
procedure Put_Point (P : in Point_Type) is
begin -- Put_Point
   Text_IO.Put ("(");
   Flt_IO.Put (P(X));
   Text_IO.Put (",");
   Flt_IO.Put (P(Y));
   Text_IO.Put (")");
end Put_Point;
```

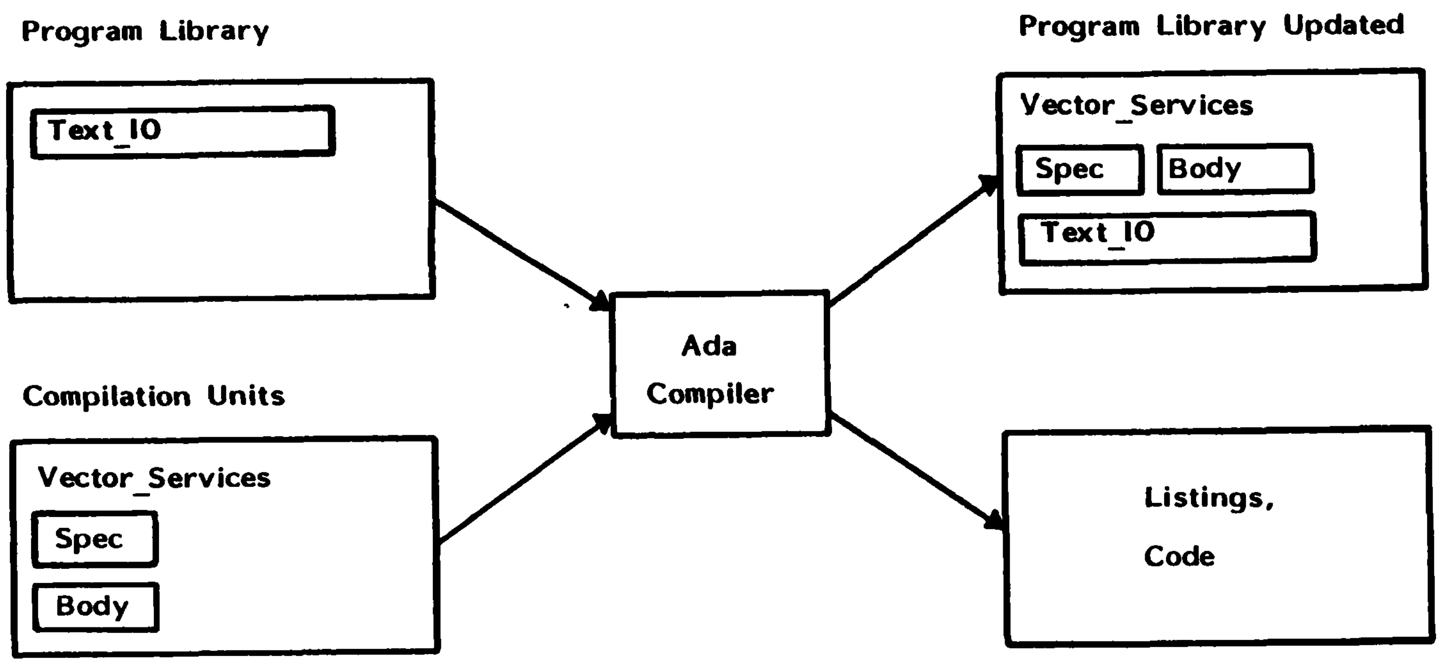
INSTRUCTOR NOTES

THE FOLLOWING EXAMPLE SPANS 3 SLIDES AND STEPS THROUGH ONE POSSIBLE WAY TO SEPARATELY COMPILE THE SYSTEM WE'VE JUST SEGMENTED.

POINT OUT THAT SPECS MUST BE COMPILED BEFORE BODIES.

AN EXAMPLE OF SEPARATE COMPILATION WITH SUBUNITS:

RUN 1

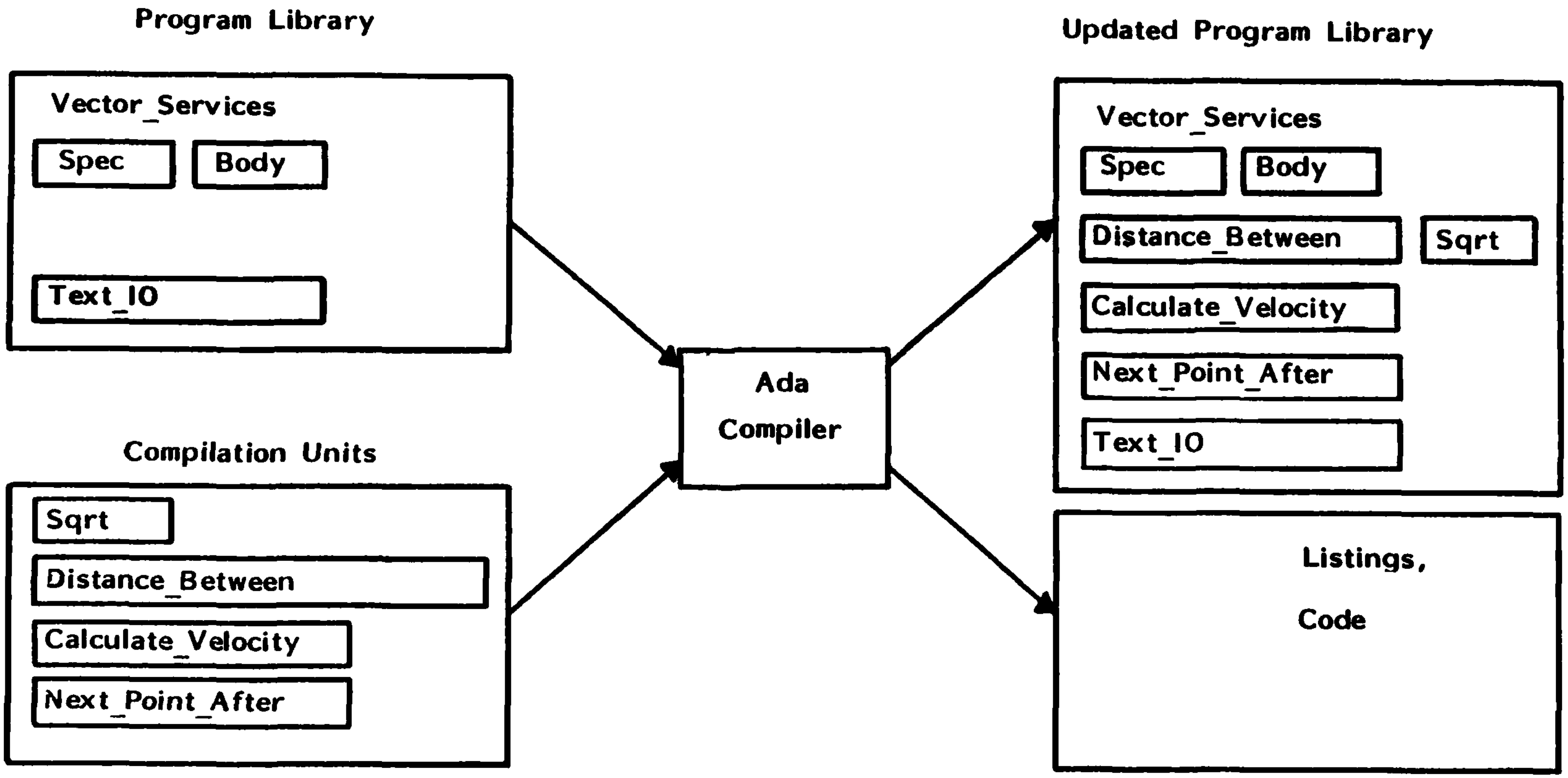


INSTRUCTOR NOTES

FOR OUR EXAMPLE, WE WILL COMPILE THE PACKAGE SUBUNITS AND ADD THEM TO THE PROGRAM LIBRARY.

POINT OUT THAT THE SUBUNITS COULD BE COMPILED INDIVIDUALLY OR IN VARIOUS OTHER COMBINATIONS, BUT WITH RESTRICTIONS ON THE ORDER - Sqrt MUST COMPILE BEFORE Distance_Between, WHICH MUST GO BEFORE Calculate_Velocity, BECAUSE OF THE DEPENDENCIES DISCUSSED EARLIER.

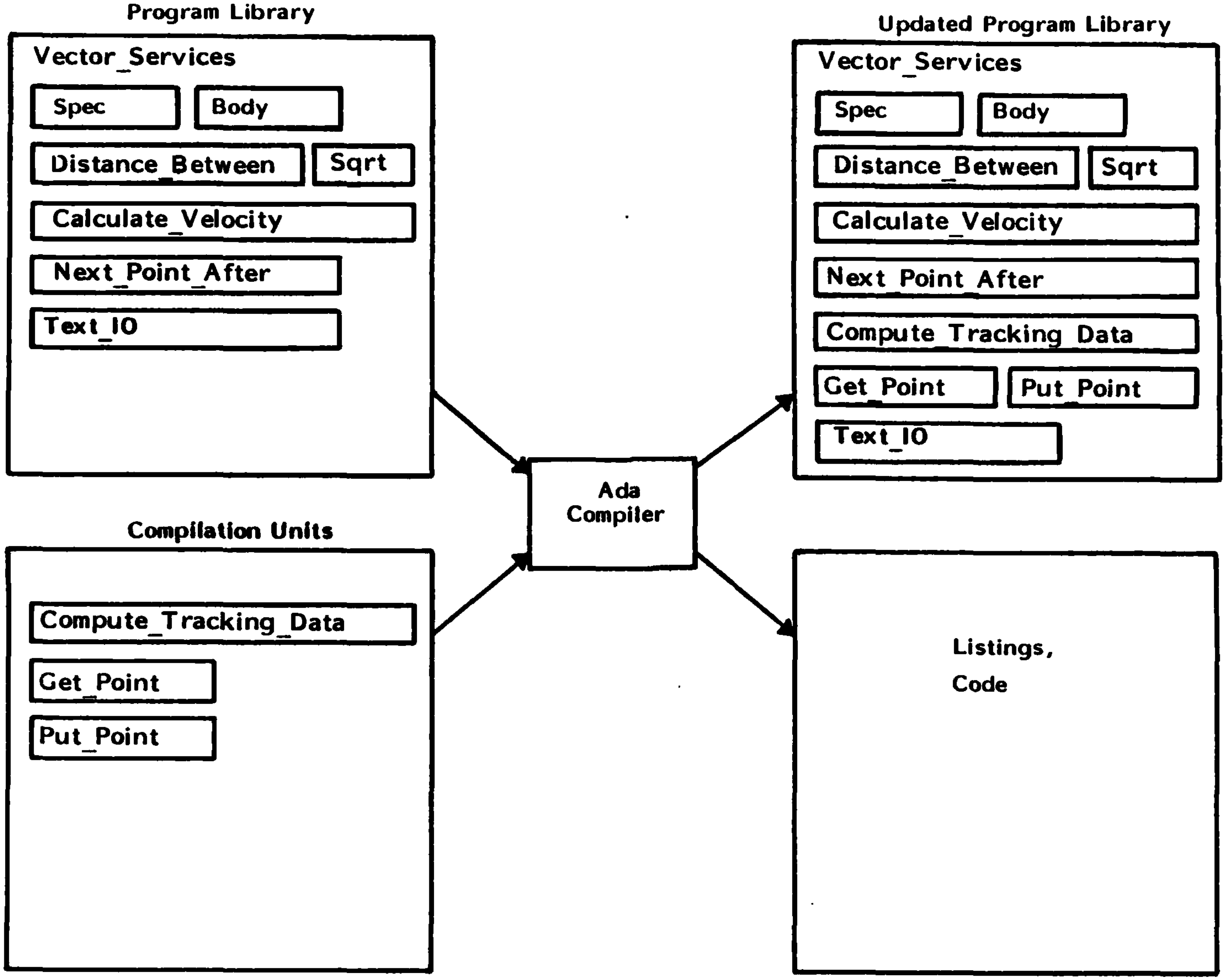
RUN 2


Program Library
Vector_Services
Spec
Body
Text_IO
Compilation Units
Sqrt
Distance_Between
Calculate_Velocity
Next_Point_After
Ada
Compiler
Updated Program Library
Vector_Services
Spec
Body
Distance_Between
Sqrt
Calculate_Velocity
Next_Point_After
Text_IO
Listings,
Code

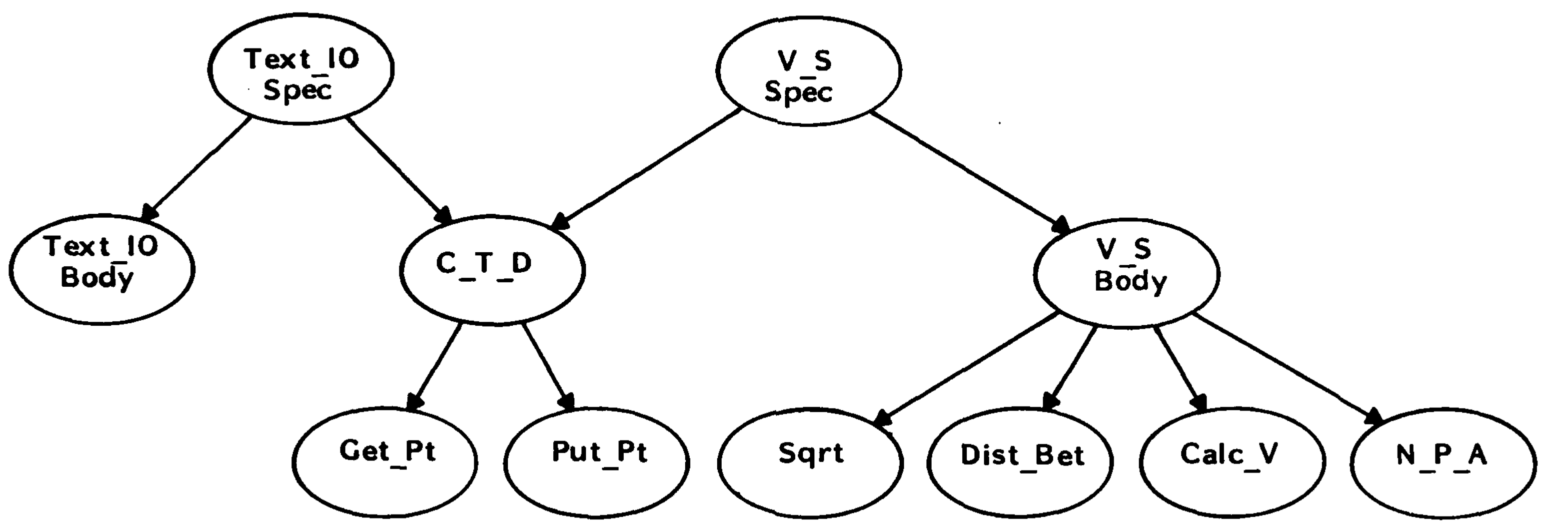
INSTRUCTOR NOTES

THE MAIN PROCEDURE COULD HAVE ALSO BEEN COMPILED ANYTIME AFTER THE PACKAGE SPECIFICATION. AGAIN, THE SUBUNITS COULD BE COMPILED SEPARATELY.

RUN 3


Program Library
Vector_Services
Spec
Body
Distance_Between
Sqrt
Calculate_Velocity
Next_Point_After
Text_IO
Compilation Units
Compute_Tracking_Data
Get_Point
Put_Point
Ada Compiler
Updated Program Library
Vector_Services
Spec
Body
Distance_Between
Sqrt
Calculate_Velocity
Next_Point_After
Compute_Tracking_Data
Get_Point
Put_Point
Text_IO
Listings,
Code

HERE IS THE DEPENDENCY DIAGRAM



ALL POSSIBLE ORDERINGS CAN BE DERIVED FROM THE ABOVE DIAGRAM.

IN-CLASS EXERCISE

SUGGEST OTHER COMPILATION

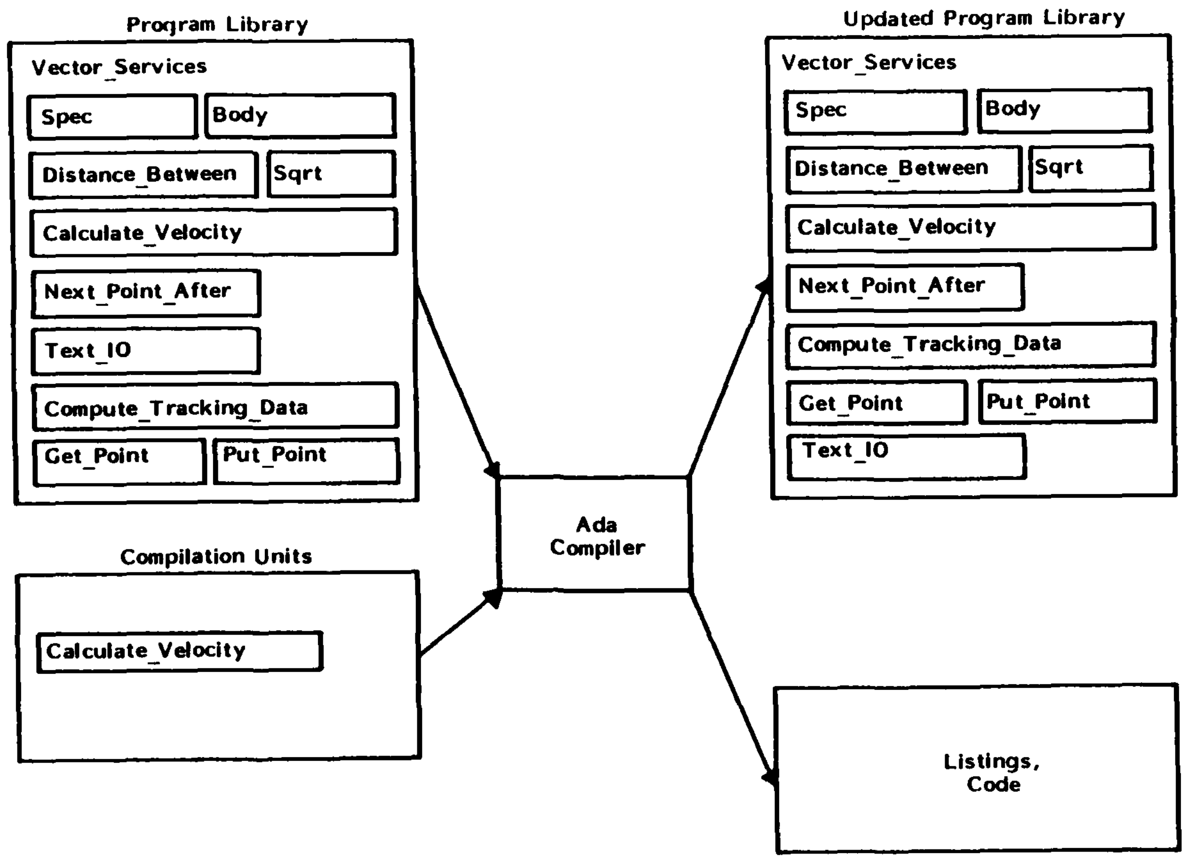
ORDER POSSIBILITIES

INSTRUCTOR NOTES

NOTE, HOW WE REDUCE THE AMOUNT OF MODIFICATION AND RECOMPILING OF THE SYSTEM. ALSO SEVERAL PROGRAMMERS COULD BE WORKING SIMULTANEOUSLY.

CHANGES TO THE SYSTEM: A SUBUNIT

WE MODIFY ONE FUNCTION, Calculate_Velocity, IN THE PACKAGE BODY.



INSTRUCTOR NOTES

THIS SECTION PRESENTS THE DESIGN CRITERIA FOR THE Ada LANGUAGE AND A GENERAL OVERVIEW OF THE FEATURES AND CONSTRUCTS THAT MAKE UP THE LANGUAGE TO PROVIDE A "FEEL" FOR THE SCOPE OF THE FEATURES AVAILABLE IN THE LANGUAGE.

AT THIS POINT ASK THE STUDENTS WHO THEY ARE, WHAT THEY DO, AND WHAT PROGRAMMING LANGUAGE THEY ARE MOST FAMILIAR WITH. (THIS LETS THE INSTRUCTOR AND STUDENTS BECOME ACQUAINTED. THE INSTRUCTOR CAN THUS ASSESS THE CLASS BACKGROUND.)

TOPIC OUTLINE

BACKGROUND AND RATIONALE FOR Ada

WRITING AN Ada PROGRAM FROM BEGINNING TO END

SUMMARY OF Ada FEATURES

INSTRUCTOR NOTES

THE FIRST THREE LANGUAGE REQUIREMENTS FROM THE STEELMAN DOCUMENT ARE GIVEN. (OTHERS ARE EFFICIENCY, SIMPLICITY, IMPLEMENTATION. THESE LAST THREE COULD BE QUITE CONTROVERSIAL AS TO WHETHER Ada ACTUALLY SATISFIES ITS OWN REQUIREMENTS.)

LIST IS IN ORDER OF IMPORTANCE OF DESIGN CRITERIA. SHOULD NOTE THAT RELIABILITY IS MORE IMPORTANT THAN EFFICIENCY. ALSO THAT READABILITY IS MORE IMPORTANT THAN WRITABILITY - A PROGRAM IS READ MANY MORE TIMES IN ITS LIFE TIME THAN IT IS WRITTEN.

MODERN SOFTWARE ENGINEERING PRINCIPLES INCLUDE MODULARITY, ABSTRACTION, LOCALIZATION, HIDING, UNIFORMITY, COMPLETENESS, CONFIRMABILITY. INSTRUCTOR SHOULD BE FAMILIAR WITH THESE CONCEPTS. (EXCELLENT REFERENCE "IF" INSTRUCTOR NEEDS THIS BACKGROUND: 'SOFTWARE ENGINEERING: PROCESS, PRINCIPLES, AND GOALS,' D.T. ROSS, J.B. GOODENOUGH, C.A. IRVINE, COMPUTER, May 1975.)

THE Ada LANGUAGE WAS DESIGNED FOR

- GENERALITY
  - MEETS A WIDE SPECTRUM OF NEEDS
- RELIABILITY
  - PROVIDES COMPILE-TIME DETECTION OF CODING ERRORS
  - ENCOURAGES MODERN SOFTWARE ENGINEERING PRINCIPLES
- MAINTAINABILITY
  - READABILITY IS MORE IMPORTANT THAN WRITABILITY
  - ENCOURAGES DOCUMENTATION
- MACHINE INDEPENDENCE
  - IMPLEMENTATION DEPENDENT LANGUAGE FEATURES CLEARLY IDENTIFIED

INSTRUCTOR NOTES

ECS NEEDS PARALLEL PROCESSING, REAL-TIME CONTROL, ERROR HANDLING, UNIQUE I/O CONTROL. GENERALLY DEALING WITH SYSTEMS THAT ARE LARGE, WILL BE IN EXISTENCE FOR MANY YEARS, UNDERGOING CONTINUAL MODIFICATIONS. RELIABILITY AND SIZE CONSTRAINTS ARE CRITICAL FACTORS IN MOST ECS. (E.G. YOU CAN'T AFFORD TO HAVE AN ERROR IN SOFTWARE NUCLEAR MISSILE).

AS A RESULT NOTE THAT REAL-TIME SYSTEM *PROCESSING, SEPARATE COMPILATION FACILITIES FOR* LARGE SYSTEM DEVELOPMENT AND EARLY ERROR DETECTOR WERE STRESSED. ALSO, SOFTWARE ENGINEERING METHODS AND PRINCIPLES SUCH AS STRONG-TYPING, ABSTRACTION, HIDING, STRUCTURED PROGRAMMING WERE EMPHASIZED AS REQUIREMENTS FOR A LANGUAGE.

DoD LANGUAGE REQUIREMENTS

	• STRONG TYPING
SOFTWARE ENGINEERING	• DATA ABSTRACTION AND INFORMATION HIDING
	• STRUCTURED CONTROL CONSTRUCTS
	• CONCURRENT PROCESSING
EMBEDDED COMPUTER SYSTEMS	• ERROR HANDLING
	• MACHINE REPRESENTATION FACILITIES
LARGE SYSTEM DEVELOPMENT	• SEPARATE COMPILATION AND LIBRARY MANAGEMENT
REUSABLE SOFTWARE	• GENERIC DEFINITION

INSTRUCTOR NOTES

A LIST OF THE FOUR STRUCTURAL BUILDING BLOCKS OF ANY Ada SYSTEM.

BRIEFLY SAY WHAT EACH DOES IN Ada, E.G. PACKAGES PROVIDE A MEANS TO COLLECT RELATED DATA AND ALGORITHMS, SUBPROGRAMS ARE SIMILAR TO OTHER LANGUAGES. THEY PROVIDE ALGORITHMS, AND TASK PROVIDES MECHANISMS FOR REAL-TIME PROCESSING.

IT IS NOT IMPORTANT FOR THE STUDENTS TO BE AWARE OF THE EXACT NATURE OF PROGRAM UNIT COMBINATIONS, JUST THAT IT <u>CAN BE DONE</u> (E.G., A TASK INSIDE A SUBPROGRAM INSIDE A PACKAGE).

PROGRAM UNITS

Ada SYSTEMS CAN CONSIST OF COMBINATIONS OF:

- PACKAGES
- SUBPROGRAMS
  - PROCEDURES
  - FUNCTIONS
- TASKS*
- GENERICS*

*NOT COVERED IN THIS MODULE

INSTRUCTOR NOTES

THE SEPARATION OF THE SPECIFICATION FROM THE BODY (THE WHAT FROM THE HOW) IS WHAT GIVES US THE <u>RELIABILITY</u> AND <u>MAINTAINABILITY</u> POINTS OF THE SLIDE. REALLY EXPLAIN THE SPECIFICATION AND BODY AND WHY IT IS IMPORTANT.

INTERFACE ERRORS ARE ONE OF THE MAJOR PROBLEMS IN INTEGRATING MODULES IN LARGE SYSTEMS. WITH THE SPECIFICATION INFORMATION, THE COMPILER CAN PERFORM VALIDITY CHECKS AT COMPILE-TIME RATHER THAN INTEGRATION TIME. IN OTHER WORDS, YOU CAN TEST THE INTERFACES OF THE DESIGN AS A WHOLE BEFORE CODING ANY OF THE ALGORITHMS. IT IS MORE COST EFFECTIVE TO CORRECT ERRORS AT THIS POINT THAN AT INTEGRATION AND TESTING.

SPECIFICATIONS CAN BE VIEWED AS LOGICAL INTERFACES.

PROGRAM UNIT STRUCTURE

ALL PROGRAM UNITS HAVE A SIMILAR FORM

- SPECIFICATION

  DESCRIBES WHAT THE PROGRAM UNIT DOES

  THIS INFORMATION IS 'VISIBLE' TO (CAN BE REFERENCED BY) THIS AND OTHER PROGRAM UNITS

- BODY

  DETAILS HOW THE PROGRAM UNIT IMPLEMENTS AN ALGORITHM OR STRUCTURE

  THIS INFORMATION IS 'HIDDEN' FROM (<u>CANNOT</u> BE DIRECTLY REFERENCED BY) OTHER PROGRAM UNITS

RELIABILITY INCREASED BECAUSE INTERFACE (SPECIFICATION) ERRORS CAN BE EASILY DETECTED

MAINTAINABILITY INCREASED BECAUSE CHANGES TO THE IMPLEMENTATION (BODY) CAN BE DONE WITHOUT AFFECTING USER PROGRAM UNITS

INSTRUCTOR NOTES

THIS IS ONE OF Ada'S STRONGEST FEATURES.

PACKAGES PROVIDE A MEANS TO PHYSICALLY GROUP LOGICALLY RELATED OBJECTS AND OPERATIONS IN SUCH A WAY THAT WHEN WE NEED TO CHANGE PORTIONS OF A SYSTEM WE CAN KNOW THE EXACT AREAS THAT WILL BE AFFECTED. THUS WE CAN REDUCE THE AFFECTED AREA TO A MINIMUM. THIS ALLOWS US CONTROL OF THE PROVERBIAL "RIPPLE EFFECT" ASSOCIATED WITH SYSTEM CHANGES.

PACKAGE

- IS BASIC STRUCTURING UNIT
- GROUPS FUNCTIONALLY RELATED DATA AND PROGRAM UNITS (ENCAPSULATION)
- PROVIDES FOR REUSABLE SOFTWARE COMPONENTS
- INCREASES MAINTAINABILITY BECAUSE EFFECT OF CHANGES CAN BE LOCALIZED

INSTRUCTOR NOTES

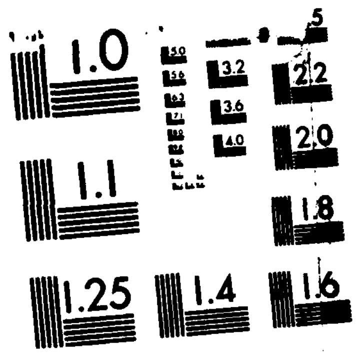
SUBPROGRAMS ARE BASICALLY AS IN OTHER LANGUAGES. SUBPROGRAMS HAVE PARAMETERS AS IN OTHER LANGUAGES.

BACKGROUND NOTE:

THESE PARAMETERS PASS VALUES (THIS WILL BE CONTRASTED LATER WITH GENERICS WHICH CAN PASS TYPES). THIS DOES NOT SAY THAT ACTUAL PARAMETERS ARE PASSED BY VALUE. IN FACT THEY CAN BE PASSED BY VALUE, VALUE RESULT OR REFERENCE.

AD-A166 366 ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA PROGRAMMING L202 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 3/8

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SUBPROGRAM

- BASIC EXECUTABLE PROGRAM UNIT
- TWO FORMS OF SUBPROGRAMS
  - PROCEDURE
  - FUNCTION

INSTRUCTOR NOTES

TASKS PROVIDE EXPRESSION OF REAL-TIME PROCESSING IN A HIGH ORDER LANGUAGE (HOL).

RENDEZVOUS PROVIDES SYNCHRONIZATION <u>AND</u> THE EXCHANGE OF DATA.

TASK

- PARALLEL THREADS OF CONTROL
- CONCURRENCY REAL WITH MULTIPROCESSORS
- CONCURRENCY APPARENT WITH SINGLE PROCESSOR
- MECHANISM FOR SYNCHRONIZATION AND DATA TRANSMISSION IS CALLED "RENDEZVOUS"

INSTRUCTOR NOTES

DON'T GO INTO THE INDIVIDUAL LISTS OF STATEMENTS. JUST SHOW THAT STATEMENTS EXIST TO HANDLE THE LISTED AREAS OF ACTION AND CONTROL (I.E. FLOW CONTROL, BASIC ACTIONS, REAL-TIME ACTIONS, EXCEPTIONS).

NOTE THAT THIS IS ALL THE STATEMENTS THERE ARE TO LEARN IN Ada AND THAT STATEMENTS ARE SIMILAR TO OTHER LANGUAGES.

STATEMENTS PROVIDE LOGIC CONTROL OR SPECIFIC ACTIONS

FLOW CONTROL:	GOTO IF (CONDITIONAL) CASE (CONDITIONAL) LOOP & EXIT (ITERATIVE) RETURN RAISE (EXCEPTIONS)
BASIC ACTIONS:	SUBPROGRAM CALLS ASSIGNMENT
REAL-TIME ACTION:	ENTRY CALL ACCEPT ABORT DELAY SELECT
EXCEPTIONS:	RAISE
DECLARATION SCOPE:	BLOCK

INSTRUCTOR NOTES

TYPE IS CONFUSING TO MANY PEOPLE WITH ONLY A FORTRAN OR ASSEMBLY LANGUAGE BACKGROUND. SIMPLY, A TYPE IS JUST A BLUEPRINT, A DESCRIPTION OF HOW SOME OBJECT WILL BEHAVE, BUT IT DOESN'T CREATE THE OBJECT IN MEMORY. A DECLARATION THEN DOES THE ACTUAL CREATION. (NOTE THE CONNECTION OF TYPE AND DECLARATION.)

EMPHASIZE STRONG TYPING ADVANTAGES AND THE EXAMPLES (BRIEFLY). IT MAKES IT SO YOU CAN'T MIX APPLES AND ORANGES ACCIDENTALLY. IF YOU WOULD NORMALLY NOT COMBINE OBJECTS (SAY IN THE REAL WORLD) THAT LOGIC CAN BE REFLECTED IN THE LANGUAGE. (THIS IS AN IMPORTANT PART OF Ada.)

TYPES

- A BLUEPRINT TO DESCRIBE (NOT CREATE)
  - A SET OF VALUES
  - THE OPERATIONS APPLICABLE TO THOSE VALUES
- PREDEFINED AND USER-DEFINED TYPES
- STRONG TYPING ALLOWS ERROR DETECTION AT COMPILE TIME
  - THE TYPE OF A VARIABLE OR PARAMETER DOES NOT CHANGE ONCE CREATED

INSTRUCTOR NOTES

ADDITIONAL Ada FEATURES

- GENERICS
- OVERLOADING
- EXCEPTIONS
- MACHINE REPRESENTATION SPECIFICATIONS

INSTRUCTOR NOTES

IF YOUR PART OF A SYSTEM HAS SPECIFIC OR LIMITED I/O NEEDS, THEN YOU ONLY HAVE TO HAVE WHAT IS ABSOLUTELY NECESSARY TO YOUR PARTICULAR FUNCTION. YOU DON'T HAVE TO HAVE ALL POSSIBLE FORMS/FORMATS OF I/O FOR ALL POSSIBLE USES. DECREASES COMPILE OVERHEAD.

INPUT/OUTPUT

- ACCESSED THROUGH PACKAGES (PREDEFINED AND USER-DEFINED)
- USER HAS COMPLETE CONTROL OF I/O

INSTRUCTOR NOTES

A SUMMARY OF WHAT/WHERE/WHY Ada IS USEFUL

AGAIN, SOFTWARE ENGINEERING PRINCIPLES IMPLY SUCH CONCEPTS AS STRUCTURED PROGRAMMING, STRONG TYPING OF DATA, MODULARITY, ABSTRACTION, AND READABILITY.

ASSIGN CHAPTER 2 OF THE PRIMER.

EMPHASIS OF Ada

- USEFUL FOR WIDE RANGE OF APPLICATIONS
  - EMBEDDED COMPUTER SYSTEMS
  - SYSTEMS PROGRAMMING
  - REAL-TIME PROGRAMMING
  - DATA PROCESSING
- DEVELOPMENT BY *PROJECT TEAMS*
- SOFTWARE ENGINEERING PRINCIPLES ENCOURAGED AND ENFORCED
- MAINTAINABILITY AND RELIABILITY

INSTRUCTOR NOTES

ALLOCATE <u>AT</u> <u>LEAST</u> 3/4 OF AN HOUR FOR THIS SECTION.

THE OBJECTIVE OF THIS SECTION IS TO INTRODUCE Ada LEXICAL ELEMENTS.

SECTION 3
LEXICAL ELEMENTS

INSTRUCTOR NOTES

POINT OUT THE SPACE CHARACTER.

CHARACTER SET

UPPER CASE

A B C D E F G H I J K L M

N O P Q R S T U V W X Y Z

LOWER CASE

a b c d e f g h i j k l m

n o p q r s t u v w x y z

DIGITS

Ø 1 2 3 4 5 6 7 8 9

SPECIAL CHARACTERS

" # % & ' ( ) * + , - . /

: ; < = > _ | ! $ @ [\ ]

^ ` { } ~ ?

SPACE CHARACTER

INSTRUCTOR NOTES

MAKE THE ANALOGY THAT LEXICAL ELEMENTS ARE THE "WORDS" OF AN Ada "SENTENCE." JUST AS THE NATURE OF AN ENGLISH SENTENCE IS DETERMINED FROM ITS WORDS AND THEIR ORDER OF OCCURRENCE (E.G., DECLARATIVE STATEMENT, QUESTION), THE NATURE OF AN Ada STATEMENT (E.G., IF, CASE) IS DETERMINED FROM ITS LEXICAL ELEMENTS.

LEXICAL ELEMENTS ARE THE "BUILDING BLOCKS" OF Ada PROGRAMS.

A PROGRAM IS A SEQUENCE OF LEXICAL ELEMENTS

- IDENTIFIERS
- NUMERIC LITERALS
- CHARACTER AND STRING LITERALS
- DELIMITERS
- COMMENTS

INSTRUCTOR NOTES

USER-DEFINED IDENTIFIERS HAVE THEIR FIRST LETTERS CAPITALIZED. WHERE AN IDENTIFIER CONSISTS OF MULTIPLE WORDS, THE FIRST LETTER OF EACH WORD IS CAPITALIZED, WITH THE EXCEPTION OF PREPOSITIONS. THIS IS A CONVENTION.

POINT OUT USE OF THE UNDERLINE TO ENHANCE READABILITY.

POINT OUT THAT THEY HAVE SEEN MOST OF THESE NAMES IN SECTION 2.

POINT OUT THAT THIS LIST IS NOT MEANT TO BE ALL-INCLUSIVE. THESE ENTITIES LISTED (E.G., PACKAGE, FUNCTION, PROCEDURE, VARIABLE, LOOP PARAMETER) ARE JUST A FEW OF THE Ada ENTITIES THAT CAN BE NAMED.

IDENTIFIERS

- USED AS NAMES

```
Vector_Services      -- A PACKAGE NAME
Sqrt                 -- A FUNCTION NAME
Calculate_Velocity   -- A PROCEDURE NAME
Distance             -- A VARIABLE NAME
Index                -- A LOOP PARAMETER NAME
```

INSTRUCTOR NOTES

POINT OUT THAT THESE ARE THE RULES FOR FORMING THEIR <u>OWN</u> IDENTIFIERS.

POINT OUT CONSEQUENCES OF FIFTH BULLET:

1. TWO UNDERSCORES IN A ROW ARE FORBIDDEN.
2. UNDERSCORES ARE FORBIDDEN AT THE BEGINNING OR END OF THE IDENTIFIER.

IDENTIFIERS

- MUST START WITH A LETTER
- SUBSEQUENT CHARACTERS ARE LETTERS, DIGITS AND UNDERSCORES
- ALL CHARACTERS ARE SIGNIFICANT, INCLUDING UNDERSCORES
- NO EMBEDDED SPACES
- AN UNDERSCORE IS ONLY ALLOWED BETWEEN TWO NON-UNDERSCORES
- UPPER AND LOWER CASE ARE CONSIDERED THE SAME
- CAN BE AS LONG AS THE ENTIRE LINE

INSTRUCTOR NOTES

"IDENTIFIERS SHOULD BE CHOSEN WITH CLARITY TO THE READER UPPERMOST IN MIND. AVOID SHORT, CRYPTIC NAMES THAT DON'T AID THE READER'S UNDERSTANDING; ON THE OTHER HAND DON'T USE LONG AND CUMBERSOME IDENTIFIERS THAT ARE HARD FOR THE EYE TO FOLLOW AND MAKE A PROGRAM HARD TO FORMAT. WHEREVER POSSIBLE SELECT NAMES PERTINENT TO THE APPLICATION."

EXAMPLES OF IDENTIFIERS

```
Calculate_Velocity          -- UNDERSCORE IS SIGNIFICANT

CalculateVelocity           -- NOT THE SAME AS Calculate_Velocity

Distance

NUMBER_OF_ITEMS             -- NO DISTINCTION MADE BETWEEN
Number_of_Items             -- UPPER AND LOWER CASE

Size_30                     -- IDENTIFIER MAY INCLUDE DIGITS

Extended_Security_Classification_Variant_Record_Type
                            -- A VERY LONG IDENTIFIER
```

INSTRUCTOR NOTES

POINT OUT THAT RESERVED WORDS CANNOT BE USED AS USER SUPPLIED IDENTIFIERS. FOR EXAMPLE:

```
procedure Delay;
     and
Delta : Float;
```

ARE ILLEGAL.

POINT OUT THAT BY CONVENTION, RESERVED WORDS ARE ALWAYS LOWER CASE. STRESS THAT IT IS CONVENTION.

POINT OUT A FEW OF THE RESERVED WORDS THAT HAVE A HIGH PRIORITY OF BEING CHOSEN FOR USER DEFINED IDENTIFIERS SUCH AS

ACCEPT (MIGHT BE USED FOR A PROCEDURE NAME)
RAISE (MIGHT BE USED FOR A PROCEDURE NAME)
ACCESS (MIGHT BE USED FOR A PROCEDURE NAME)

TO ALERT THE STUDENT THAT THESE MAY <u>NOT</u> BE USED.

IDENTIFIERS

- INCLUDE RESERVED WORDS

- RESERVED WORDS MAY BE WRITTEN IN EITHER LOWER CASE OR UPPER CASE

  (CONVENTION IS TO WRITE IN LOWER CASE)

abort	declare	generic	of	select
abs	delay	goto	or	separate
accept	delta		others	subtype
access	digits	if	out	
all	do	in		task
and		is	package	terminate
array	else		pragma	then
at	elsif		private	type
	end	limited	procedure	
begin	entry	loop		
body	exception		raise	use
	exit		range	
		mod	record	when
			rem	while
		new	renames	with
case	for	not	return	
constant	function	null	reverse	xor

INSTRUCTOR NOTES

STRESS THAT THE UNDERSCORE IS NOT SIGNIFICANT IN NUMERIC LITERALS AND THAT THIS IS DIFFERENT FROM THE RULE FOR IDENTIFIERS.

UNDERSCORES CAN BE USED IN THE SAME WAY COMMAS ARE USED IN ORDINARY ENGLISH TO MAKE IT EASY TO READ LONG NUMBERS.

POINT OUT THAT THE UNDERSCORE MUST HAVE A DIGIT ON BOTH SIDES. FOR EXAMPLE 3_.14 IS ILLEGAL.

MENTION THAT THE Ada LRM REFERS TO THE UNDERSCORE AS UNDERLINE.

NUMERIC LITERALS

- INTEGER LITERALS
  - --- NO DECIMAL POINT
- REAL LITERALS
  - --- HAVE A DECIMAL POINT

ISOLATED UNDERSCORE CHARACTERS MAY ONLY BE INSERTED BETWEEN ADJACENT DIGITS TO FACILITATE READABILITY, BUT ARE NOT SIGNIFICANT.

INSTRUCTOR NOTES

POINT OUT THAT COMMAS ARE NOT ALLOWED IN NUMERIC LITERALS. UNDERSCORE IS USED INSTEAD TO PRESERVE READABILITY.

EXPONENTS IN INTEGER LITERALS ARE ESSENTIALLY A SHORTHAND FOR TRAILING ZEROES.

BASED NUMBERS ARE EXPLAINED IN SECTION 14. BASED NUMBERS ARE AN ALTERNATE WAY OF WRITING INTEGER NUMERALS.

INTEGER LITERALS

- CAN HAVE AN EXPONENT BUT IT MUST BE POSITIVE OR ZERO
- NO DECIMAL POINT
- ALTERNATE NOTATION FOR NON-DECIMAL INTEGER NUMERALS (FOR EXAMPLE, BINARY, OCTAL, OR HEXADECIMAL NUMERALS)

EXAMPLES:

```
2500              -- ONE WAY TO WRITE 2,500
2_500             -- ALTERNATIVE WAY TO WRITE 2,500
25E2              -- ANOTHER ALTERNATIVE WAY TO WRITE 2,500
123_45_6789       -- APPROPRIATE WAY TO WRITE A SOCIAL SECURITY NUMBER
1_234_567_890     -- APPROPRIATE WAY TO WRITE 1,234,567,890
123_456_789E1
(1200E-2 IS NOT ALLOWED)
```

INSTRUCTOR NOTES

"'RADIX POINT,' IF YOU INSIST!"

(GENERAL NAME FOR "DECIMAL POINT" WHEN USING DIFFERENT BASES)

EACH EXAMPLE IS A DIFFERENT WAY TO REPRESENT THE SAME REAL LITERAL VALUE.

REAL LITERALS

- MUST CONTAIN A DECIMAL POINT
- DECIMAL POINT MUST NOT BE FIRST OR LAST
- DECIMAL POINT MUST NOT BE PRECEDED OR FOLLOWED BY AN UNDERSCORE
- EXPONENT INDICATES THE POWER OF TEN
  THE PRECEDING NUMBER IS TO BE MULTIPLIED BY (MAY BE POSITIVE OR NEGATIVE)
- EXPONENT MUST BE AN INTEGER
- ALTERNATE NOTATION FOR NON-DECIMAL REAL NUMERALS

EXAMPLES:

```
12.75
1275.0E-2
0.1275E2
1_275.0E-2
(1275.E-2 IS NOT ALLOWED, NOR IS .1275E2)
```

INSTRUCTOR NOTES

"WHEREAS CASE OF A LETTER DOESN'T MATTER IN FORMING IDENTIFIERS, CASE DOES MATTER IN CHARACTER AND STRING LITERALS."

POINT OUT THAT DOUBLE QUOTES " SURROUND STRINGS. Put ("$"); WOULD OUTPUT THE STRING CONTAINING THE SINGLE CHARACTER $.

DRAW ATTENTION TO THE FACT THAT IF THE CHARACTER APOSTROPHE IS DESIRED AS A CHARACTER LITERAL, IT NEED NOT BE DUPLICATED ('''), WHEREAS IF THE DOUBLE QUOTE IS DESIRED IN A STRING LITERAL, IT MUST BE DUPLICATED.

CHARACTER LITERALS AND STRING LITERALS

- CHARACTER LITERALS
  - FORMED BY ENCLOSING ONE CHARACTER BETWEEN SINGLE APOSTROPHE CHARACTERS
  - ' ' AND ''' ARE VALID CHARACTER LITERALS
  - EXAMPLES:

```
with Text_IO;
procedure Output_Prompt is
begin -- Output_Prompt
   Text_IO.Put ('$'); -- WRITE THE PROMPT
end Output_Prompt;
```

- STRING LITERALS
  - SEQUENCE OF ZERO OR MORE CHARACTERS ENCLOSED IN QUOTES
  - EXAMPLES:

```
Text_IO.Put ("Welcome to Ada");
Text_IO.Put ("We claim, ""It's not that tough""");
```

INSTRUCTOR NOTES

PRONUNCIATION:

:=	"BECOMES" (Ada LANGUAGE REFERENCE MANUAL) ALSO "GETS," "IS ASSIGNED"
=>	THE PRONUNCIATION VARIES DEPENDING UPON THE CONSTRUCT WITHIN WHICH IT APPEARS. BASICALLY, IT IS KNOWN AS AN "ARROW." ITS OTHER NAMES ARE: "THEN" (AFTER when CLAUSE) "IS" (NAMED PARAMETER NOTATION) "THE FINGER" (WHEN WORKING OVERTIME ON A FRIDAY NIGHT)
<>	"BOX" (Ada LANGUAGE REFERENCE MANUAL)

DELIMITERS

SPECIAL CHARACTERS

```
&  "  (  )  *  +  ,  .  '  _
/  :  ;  <  =  >  #  -  [blank]
```

COMPOUND SYMBOLS

:=	ASSIGNMENT
..	RANGE DEFINITION
**	EXPONENTIATION OPERATION
>= <= /=	RELATIONAL OPERATORS
<< >>	IDENTIFIES STATEMENT LABELS
=>	INDICATES RELATIONSHIP BETWEEN A NAME AND A VALUE, ACTION, OR DECLARATION
<>	STANDS FOR INFORMATION TO BE FILLED IN LATER
--	COMMENT

INSTRUCTOR NOTES

POINT OUT THAT, SINCE THE COMMENT IS TERMINATED BY THE END OF THE LINE, IT CAN'T BE FOLLOWED BY ANY PROGRAM TEXT ON THE SAME LINE.

THIS IS DIFFERENT FROM PASCAL AND C, THE SAME AS ASSEMBLER.

UNLIKE FORTRAN, Ada ALLOWS OTHER ELEMENTS TO PRECEDE A COMMENT ON THE SAME LINE.

POINT OUT USE OF BLOCK COMMENT AND IN LINE COMMENT.

COMMENTS

- START WITH TWO HYPHENS AND ARE TERMINATED BY THE END OF THE LINE
- MAY NOT PRECEDE OTHER LEXICAL UNITS ON THE SAME LINE
- HAVE NO EFFECT ON THE MEANING OF THE PROGRAM.
- PURPOSE IS TO 'ENLIGHTEN' THE READER.

EXAMPLE:

```
-- Exchange First_Value and Second_Value only if
-- First_Value is greater than Second_Value
if First_Value > Second_Value then
   Temp         := First_Value;
   First_Value  := Second_Value;
   Second_Value := Temp;
end if;           -- if First_Value <= Second_Value
```

INSTRUCTOR NOTES

REFERENCE THE TRACKING PROGRAM CODE.

LEXICAL STYLE

- FREE FORMAT LANGUAGE - USE INDENTATION AND BLANK LINES FOR READABILITY.
- LEXICAL ELEMENTS MUST FIT ON ONE LINE.
- SPACES ARE OPTIONAL BETWEEN MOST LEXICAL ELEMENTS BUT MANDATORY BETWEEN TWO LEXICAL ELEMENTS WHICH WITHOUT SEPARATING SPACE COULD BE CONSTRUED AS ONE LEXICAL ELEMENT.
- NO CONTINUATION MARKS (I.E. STATEMENTS MAY CROSS LINE BOUNDARIES).
- LINE MAY CONTAIN MORE THAN ONE STATEMENT.
- STATEMENT MUST BE TERMINATED BY A SEMICOLON.

INSTRUCTOR NOTES

THE ARROWS ON THE LEFT POINT TO LEGAL VARIATIONS, ALTHOUGH NOT NECESSARILY RECOMMENDED. BLANK LINES ARE RECOMMENDED FOR READABILITY. STATEMENT CROSSING LINE BOUNDARIES ARE NOT RECOMMENDED IN ALL CASES. IT IS NOT RECOMMENDED HERE AS IT CAN FIT ON ONE LINE. IT IS ACCEPTABLE FOR A PROCEDURE OR FUNCTION DECLARATION WITH MULTIPLE PARAMETERS.

THE ARROWS ON THE RIGHT POINT TO ILLEGAL AREAS OF THE CODE.

AN EXAMPLE

```
Context:  subtype Scores_Type is Float range 0.0 .. 100.0;
            type List_Type is array (1..15) of Scores_Type;

Example:        procedure Calculate_Median
                     (List:  inList_Type <--- ** Illegal, a space is mandatory here
                      Midpoint: out Scores_Type) is

blank lines          Index_1,
                     Index_2: Positive;
                     Temp_List : List_Type := List;

                begin -- Calculate_Median

indentation -------> Index_1 := (Temp_List'Last+1)/2;
statement           {Index_2 := (Temp_List'Last/2)
crossing line ----->{             +1;
boundaries

multiple ----------> Sort(Temp_List); Midpoint:= (Temp_List(Index_1) + Temp_List(Index_2))/2.0;
statements
per line

                end Calculate_ <-------- ** illegal, identifiers may not
                Median;                     cross line boundaries
```

INSTRUCTOR NOTES

ALLOW 15-20 MINUTES FOR STUDENTS TO COMPLETE THE EXERCISE. ANSWERS TO THE QUESTIONS APPEAR BELOW.

1.	Too_Bad_This_Is_Not_An_Identifier	an identifier cannot start with a number
2.	Security_Classification_Type	missing underscore
3.	OK	
4.	Char_Count	illegal double underscore
5.	OK	
6.	-- of an incoming message	need -- on second line
7.	3.14 -- An ...	need -- before comment
8.	3.14	underscore must be followed by a number
9.	300_000	illegal, use _ for readability, not,
10.	X := 4;	illegal space between : and =. illegal : at end
11.	"CAR NAME MILES PER GALLON" or	
	"""CAR NAME"" MILES PER GALLON"	illegal quoted string in string
12.	300E2 or 300.0E-2	Integer cannot be raised to negative power

ASSIGN CHAPTER 3 OF THE PRIMER.

CLASS EXERCISE

INDICATE WHAT, IF ANYTHING, IS WRONG WITH THE FOLLOWING PROGRAM ELEMENTS, AND CORRECT THOSE IN ERROR.

```
1.    2_Bad_This_Is_Not_An_Identifier
2.    Security Classification_Type
3.    Channel_Mode
4.    CHAR__COUNT
5.    x
6.    -- The purpose of this procedure is to validate the security prosign
         of an incoming message
7.    3.14 - An abbreviated definition of PI
8.    3_.14
9.    300,000 -- speed of light (km/sec)
10.   X    : =  4:
11.   ""CAR NAME"        MILES PER GALLON"
12.   300E-2
```

INSTRUCTORS NOTES

ALLOCATE AT LEAST ONE AND ONE HALF HOURS OF LECTURE FOR THIS SECTION.

ASSIGN EXERCISES 3 AND 4 AT THE COMPLETION OF THIS SECTION.

THE OBJECTIVE OF THIS SECTION IS TO INTRODUCE Ada DATA OBJECTS, INTRODUCE THE CONCEPT OF PREDEFINED TYPES, AND OBJECTS OF THESE TYPES.

SECTION 4
INTRODUCTION TO DATA

INSTRUCTOR NOTES

POINT OUT THAT VARIABLES AND CONSTANTS ARE OBJECTS IN Ada.

POINT OUT THAT THE OBJECT DECLARATION ALLOCATES SPACE FOR THE OBJECT TO HOLD VALUES.

OBJECTS

- OBJECTS ARE CONTAINERS THAT HOLD DATA.

- IN ADDITION TO A VALUE, EVERY OBJECT HAS A PROPERTY CALLED TYPE, WHICH DEFINES THE KIND OF DATA THAT THE OBJECT MAY HOLD.

INSTRUCTOR NOTES

THE OBJECT DECLARATION ALLOCATES SPACE FOR THE OBJECT TO HOLD DATA.

AN OBJECT CAN ONLY ACCEPT VALUES THAT "FIT" THE DEFINITION FOR ITS TYPE.

OBJECTS

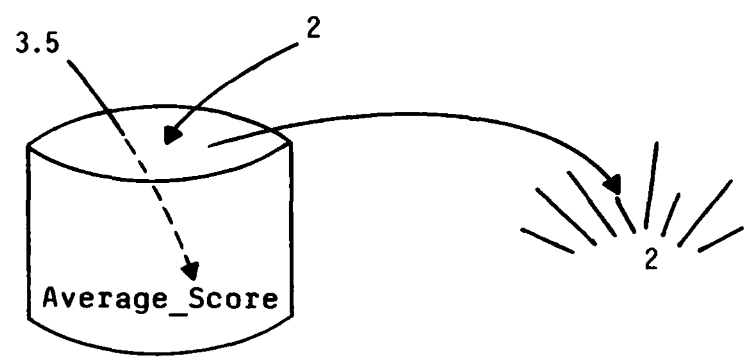
IF WE HAVE THE OBJECT DECLARATION

```
Average_Score : Float;
```

THEN BY DEFINITION: TYPE Float MUST HAVE DECIMAL POINT

Average_Score IS AN OBJECT OF TYPE Float

THEN



INSTRUCTOR NOTES

OBJECTS

- ALL OBJECTS MUST BE DECLARED IN Ada

- OBJECTS ARE DECLARED IN THE DECLARATIVE PART

```
with Text_IO; use Text_IO;
procedure Echo is

    Number : Integer;                                  } DECLARATIVE PART
    package Int_IO is new Integer_IO (Integer);
    use Int_IO;

begin -- Echo

    Get (Number);                                      } EXECUTABLE PART
    Put (Number);

end Echo;
```

INSTRUCTOR NOTES

"THIS IS IMPORTANT!"

POINTS ABOUT OTHER LANGUAGES:

- FORTRAN, JOVIAL, PASCAL: BASICALLY THE SAME IDEA
- ASSEMBLER: THINGS LIKE APPLYING A CHARACTER OPERATION TO AN INTEGER ARE NOT ALLOWED IN Ada

"INTEGER" IS A TYPE WHICH NAMES A SET OF VALUES.

"2" IS A LITERAL WHICH REPRESENTS A VALUE.

BE CAREFUL OF THE WORD "SET". IT HAS A DEFINITE MEANING IN PASCAL. IF CONFUSION ARISES, TELL THEM TO THINK OF IT AS A "GROUP".

TYPES

- SOME TYPES ARE GIVEN TO US AS PART OF THE LANGUAGE
- SOME TYPES WILL BE USER-DEFINED
- A TYPE IS CHARACTERIZED BY:
  - A SET OF VALUES
  - A SET OF OPERATIONS APPLICABLE TO THOSE VALUES

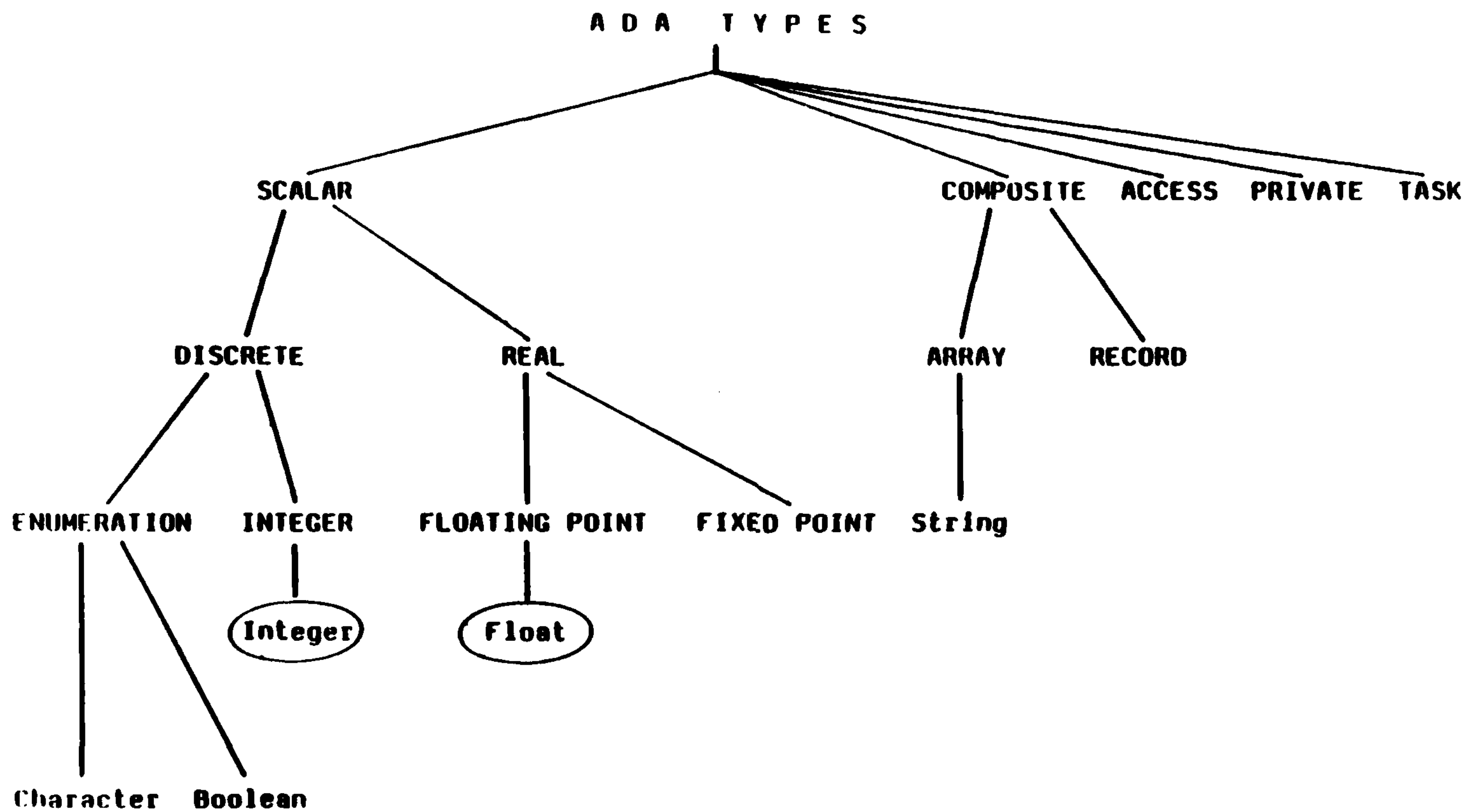
INSTRUCTOR NOTES

"Ada SUPPORTS A WIDE VARIETY OF DATA TYPES, WHICH WILL BE DISCUSSED AT VARIOUS POINTS IN THIS MODULE. THESE TYPES ARE CLASSIFIED AS BELONGING TO VARIOUS CATEGORIES. THE TREE ON THE SLIDE ILLUSTRATES SOME OF THESE CATEGORIES. IN THIS SECTION WE WILL CONSIDER THE TWO TYPES BEST KNOWN TO PROGRAMMERS IN OTHER LANGUAGES, NAMELY Integer AND Float, AND LAY THE FOUNDATIONS FOR AN UNDERSTANDING OF Ada TYPES IN GENERAL. WE'LL FILL IN THE TREE AS WE GO ALONG."

IF NECESSARY, EXPLAIN Integer AND Float FOR THE BENEFIT OF STUDENTS LACKING, OR WEAK IN, MODULE PREREQUISITES. (Float IS KNOWN AS Real IN MOST OTHER LANGUAGES.)

IN GENERAL, THIS SECTION WILL USE PREDEFINED TYPES IN OBJECT DECLARATIONS.

THE PREDEFINED TYPES Integer AND Float



REAL LITERALS CAN BE USED AS VALUES OF TYPE Float, BUT NOT OF TYPE Integer. Integer LITERALS CAN BE USED AS VALUES OF TYPE Integer, BUT NOT OF TYPE Float. (REMEMBER THE CONTAINER!)

INSTRUCTOR NOTES

POINT OUT AGAIN

~~REPEAT AD NAUSEUM~~ THAT TYPES ARE DEFINED BY A SET OF VALUES THAT OBJECTS OF THE TYPE MAY ACQUIRE AND A SET OF OPERATIONS THAT CAN BE PERFORMED ON THOSE OBJECTS.

PREDEFINED TYPE INTEGER

<u>SET OF VALUES</u>

IMPLEMENTATION DEPENDENT

<u>SET OF OPERATIONS</u>

OPERATORS	RESULT TYPE
<u>Numeric</u>	
+ - * / ** mod rem	Integer
<u>Relational</u>	
= /= < <= > >=	Boolean
<u>Membership</u>	
in not in	Boolean
<u>Unary</u>	
+ - abs	Integer

INSTRUCTOR NOTES

POINT OUT AGAIN

~~REPEAT AD NAUSEUM~~ THAT TYPES ARE DEFINED BY A SET OF VALUES THAT OBJECTS OF THE TYPE MAY ACQUIRE AND A SET OF OPERATIONS THAT CAN BE PERFORMED ON THOSE OBJECTS.

PREDEFINED TYPE FLOAT

SET OF VALUES

IMPLEMENTATION DEPENDENT

SET OF OPERATIONS

OPERATORS	RESULT TYPE
Numeric	
+ - * / **	Float
Relational	
= /= < <= > >=	Boolean
Unary	
+ - abs	Float

INSTRUCTOR NOTES

WHY ONLY A POSITIVE EXPONENT FOR INTEGERS? IF YOU HAD 2^{-2} IT WOULD BE 1/4!

EXPONENTIATION

- INTEGER VALUES AND OBJECTS CAN ONLY BE RAISED TO POSITIVE INTEGER EXPONENT.
- FLOAT VALUES AND OBJECTS CAN ONLY BE RAISED TO POSITIVE OR NEGATIVE INTEGER EXPONENT.

INSTRUCTOR NOTES

POINT OUT THAT HIERARCHY IS SIMILAR TO OTHER LANGUAGES LIKE FORTRAN AND PASCAL

POINT OUT THAT

a * b / c * d is $\frac{abd}{c}$

WHEREAS

(a * b) / (c * d) is $\frac{ab}{cd}$

HIERARCHY OF OPERATORS

Highest Precedence

↓

Lowest Precedence

parentheses	( )				
highest precedence	**	abs	not		
multiplying	*	/	mod	rem	
unary	+	-			
binary	+	-	&		
relational/membership	=	/=	<	<=	>
	>=	in	not in		
logical	and	or	xor	and then	
	or else				

INSTRUCTOR NOTES

"WHAT OTHER LANGUAGES (LIKE FORTRAN OR ASSEMBLY LANGUAGES) REFER TO AS 'CONSTANTS' ARE CALLED 'LITERALS' IN Ada, E.G.,

```
273
  9.25
1.457E-21
  1
```

TWO KINDS OF OBJECTS

- VARIABLES
  - HAVE NAMES GIVEN IN VARIABLE DECLARATIONS
  - CAN BE ASSIGNED VALUES DURING THE PROGRAM
- CONSTANTS
  - HAVE NAMES GIVEN IN CONSTANT DECLARATIONS
  - INITIALIZED AT THE BEGINNING OF A PROGRAM UNIT
  - CANNOT BE CHANGED DURING THE EXECUTION OF THE PROGRAM UNIT
  - A CONTAINER WHOSE CONTENTS NEVER CHANGE.
  - 2.6 IS NOT A "CONSTANT," BUT A "LITERAL"

INSTRUCTOR NOTES

POINT OUT WHICH IS OBJECT, WHICH IS TYPE, AND WHICH IS INITIAL VALUE IN THE EXAMPLES.

OCCASIONALLY, SOME ~~SMART ALECK~~ ~~TROUBLEMAKER~~ ADVANCED STUDENT WILL ~~BADGER YOU WITH~~ ASK THE ~~POINTLESS AND OBSCURE~~ USEFUL AND INTERESTING QUESTION:

CAN I USE THE NAME OF A TYPE AS AN IDENTIFIER FOR AN OBJECT, E.G.,

```
Float : Integer;
```

ANSWER:

1. A COMPLETE ANSWER IS BEYOND THE SCOPE OF THIS SECTION (AND OF THIS MODULE).

2. GENERALLY SPEAKING, YES, BUT IT WOULD BE BAD PRACTICE FOR TWO REASONS:

   a. THERE ARE MANY SITUATIONS, DEPENDING ON, AMONG OTHER THINGS, <u>WHERE</u> YOU MAKE THIS DECLARATION, WHICH <u>WOULD</u> MAKE THIS ILLEGAL.

   b. A MORE IMPORTANT REASON IS CLARITY TO HUMAN READERS -- ONE OF THE MAIN PRINCIPLES BEHIND THE DESIGN OF Ada IS READABILITY, AND SUCH PRACTICES DEFEAT THAT GOAL.

3. IF YOU WANT TO KNOW MORE ABOUT THIS FASCINATING TOPIC, LET'S DISCUSS IT AFTER CLASS. ~~IN OTHER WORDS, STOP WASTING EVERYONE'S TIME WITH OBSCURE QUESTIONS ABOUT STUPID PRACTICES AND SIT DOWN, YOU COMMUNIST!~~

VARIABLE DECLARATIONS

SYNTAX:

```
Name {,Name}  :   Type_Name [ := Initial_Value];
```

Note: Use of curly brackets { } indicates optional repetition

Use of square brackets [ ] indicates optional part of declaration

EXAMPLES:

```
Intelligence_Quotient             : Integer := 2;
Number                            : Integer;
Velocity, Altitude, Acceleration  : Float;
Temperature                       : Float := 273.0;
Velocity, Altitude, Acceleration  : Float := 0.0;         -- Legal
Value_1, Value_2, Value_3         : Integer := 1, 2, 3;   -- **Illegal
```

INSTRUCTOR NOTES

POINT OUT THAT IT IS A MECHANISM WHICH PROHIBITS OBJECTS FROM OBTAINING VALUES OUTSIDE THE RANGE CONSTRAINT SUPPLIED.

RANGE CONSTRAINTS FOR VARIABLES

- PLACES UPPER AND LOWER BOUNDS ON VALUES THAT THE OBJECTS MAY ACQUIRE.

SYNTAX:

```
Variable_Name : Type_Name [range L .. R];
```

or

```
Variable_Name : Type_Name [range L .. R] := Initial_Value;
```

EXAMPLE:

```
Cruising_Altitude_In_Feet : Integer range 15000 .. 35000;
```

INSTRUCTOR NOTES

POINT OUT THAT ANY ATTEMPT TO ASSIGN A VALUE TO THE OBJECT WHICH IS OUTSIDE THE RANGE CONSTRAINT WILL CAUSE THE PROGRAM TO CRASH. EXPLAIN THAT Ada HAS A MECHANISM (CALLED EXCEPTIONS) WHICH WILL ALLOW US TO PROGRAM FOR CONTINUED EXECUTION AND AVOID A CRASH. EXCEPTIONS ARE COVERED LATER IN L202.

RANGE CONSTRAINT

range L .. R

WHERE

- range IS A RESERVED WORD
- L REPRESENTS THE LOWER BOUND
- R REPRESENTS THE UPPER BOUND

IF AN OBJECT ACQUIRES A VALUE WHICH IS <L OR >R THEN SUSPENSION OF NORMAL PROGRAM EXECUTION OCCURS. FOR NOW, IT SUFFICES TO SAY THAT VALUES NOT SATISFYING A SPECIFIC RANGE CONSTRAINT CANNOT BE APPLIED TO OBJECTS THAT HAVE BEEN DECLARED WITH THAT RANGE CONSTRAINT.

INSTRUCTOR NOTES

THE POINT IS THAT ANY ATTEMPT TO ASSIGN A VALUE OUTSIDE THE RANGE CONSTRAINT WILL CAUSE THE PROGRAM TO CRASH.

EXCEPTIONS CAN HANDLE THIS SITUATION BUT ARE DISCUSSED LATER.

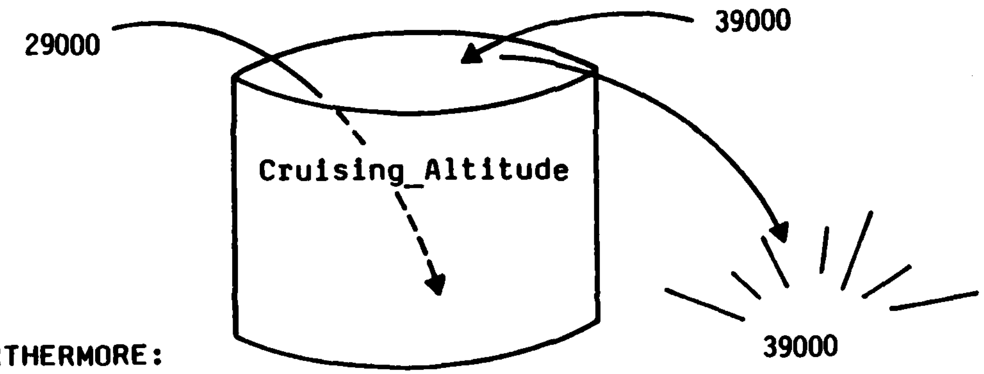
THE RANGE CONSTRAINT MECHANISM CAN BE USED VERY EFFECTIVELY TO ALLOW THE COMPILER TO CHECK THAT ONLY CERTAIN VALUES CAN BE ASSIGNED TO CERTAIN VARIABLES.

RANGE CONSTRAINT

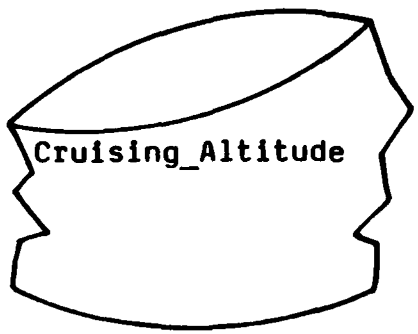
IF WE HAVE

```
Cruising_Altitude : Integer range 15000 .. 35000;
```

THEN



AND FURTHERMORE:



INSTRUCTOR NOTES

REMIND STUDENTS AGAIN THAT WHAT OTHER LANGUAGES REFER TO AS "CONSTANTS" Ada REFERS TO AS "LITERALS" (E.G. 273, 9.25).

Ada CONSTANTS HAVE NAMES, LIKE VARIABLES, BUT THEIR VALUES CANNOT CHANGE (WITHIN THE PROGRAM UNIT DECLARING THEM). MOREOVER, THEY ARE <u>EXPLICITLY PREVENTED</u> FROM ASSUMING NEW VALUES. (THE COMPILER WON'T ALLOW IT.)

Ada CONSTANTS ARE REALLY "READ-ONLY" VARIABLES.

POINT OUT THAT A DECLARATION CAN REFER TO CONSTANTS THAT OCCURRED IN A PREVIOUS DECLARATION.

POINT OUT ABSENCE/PRESENCE OF DECIMAL POINTS FOR INTEGER/REAL LITERALS.

CONSTANT DECLARATIONS

SYNTAX:

```
Name {,Name}  :  constant Type_Name  := Value;
```

EXAMPLES:

```
Pi             :  constant Float := 3.14159;
Length, Width  :  constant Float := 5.2;
Area           :  constant Float  := Length * Width;
Days_in_April  :  constant Integer := 30;
```

THERE MUST BE ONE (AND ONLY ONE) VALUE PER LINE, E.G.:

```
Length, Width : constant Float := 0.0;       -- Legal
Length, Width : constant Float := 5.2, 3.8; -- ** ILLEGAL
```

INSTRUCTOR NOTES

1. MORE READABLE BECAUSE CONSTANT NAME GIVES INFORMATION AS TO ITS PURPOSE, ORIGIN, USE, ETC. NAMES LIKE Initial_Velocity, Pi, Atmospheric_Pressure, Interest_Rate ARE HELPFUL; NAMES LIKE Twenty (:= 20) ARE NOT.

2. DON'T OVERDO IT: N := N + 1 IS OKAY

3. BOTTOM LINE IS READABILITY: NAMES SHOULD BE APPROPRIATE TO APPLICATION.

ADVANTAGES OF SYMBOLIC CONSTANTS

- READABILITY
- MAINTAINABILITY

  only need to change one time to change a value throughout the program

INSTRUCTOR NOTES

POINT OUT THAT AS WRITTEN, IF MORE PRECISION IS DESIRED, THE VALUE FOR Pi NEED ONLY BE CHANGED ONCE, IN THE DECLARATIVE PART. THERE IS NO NEED TO SEARCH THROUGH THE EXECUTABLE CODE FOR EACH OCCURRENCE OF 3.14159 AND POSSIBLY MISS ONE.

IF A STUDENT REMARKS, "I CAN CODE THAT QUICKER IN MY LANGUAGE," REMIND HIM/HER THAT Ada WAS DESIGNED TO HELP IN THE MAINTENANCE PHASE. Ada WAS DESIGNED FOR THE READERS. THE EXTRA TIME REQUIRED UP FRONT TO WRITE Ada CODE WILL PRODUCE MORE MAINTAINABLE CODE.

STYLEWISE THESE TWO ASSIGNMENT STATEMENTS WOULD BE BETTER CODED AS FUNCTIONS IN AN Ada PACKAGE. OUR STUDENTS AT THIS POINT ARE NOT SOPHISTICATED ENOUGH TO UNDERSTAND THAT. IF, HOWEVER, SOMEONE DOES MAKE A REMARK, STATE THAT THE POINT HERE IS TO ADDRESS NAMED CONSTANTS ON ONE SLIDE.

ADVANTAGES OF SYMBOLIC CONSTANTS

```
with Text_IO; use Text_IO;
procedure Calculate_Circle_Stuff (Radius : in Float) is

     Pi                        : constant Float := 3.14159;
     Area_Of_Circle            : Float;
     Circumference_Of_Circle : Float;
     package FL_IO is new Float_IO (Float);
     use FL_IO;

begin -- Calculate_Circle_Stuff

     Area_Of_Circle            := Pi * Radius ** 2;
     Circumference_Of_Circle := 2.0 * Pi * Radius;
     Put (Area_Of_Circle);
     Put (Circumference_Of_Circle);

end Calculate_Circle_Stuff;
```

INSTRUCTOR NOTES

"THE IMPORTANCE OF STRONG TYPING WILL BECOME MORE APPARENT AS WE INTRODUCE OTHER DATA TYPES.

THE REASONS ARE UNRELATED TO THE ORIGINAL, IMPLEMENTATION-DEPENDENT CONSIDERATIONS THAT 'INTEGER ARITHMETIC IS FASTER'."

ASSIGNMENT AND STRONG TYPING

- THE ASSIGNMENT OPERATOR IN Ada IS :=
- ASSIGNMENT STATEMENTS MUST BE TYPE COMPATIBLE
- VALUES OF ONE TYPE CANNOT BE ASSIGNED TO VARIABLES OF A DIFFERENT TYPE
  - PROGRAM REFLECTS ABSTRACT LOGICAL ENTITIES OF THE PROBLEM
  - COMPILER DETECTS ERRORS CAUSED BY TYPE MISMATCH (BETWEEN VARIABLE AND VALUE)
  - COMPILER CHECKS FOR SOFTWARE CONSISTENCY
- IN MOST ARITHMETIC OPERATIONS, THE VALUES TO WHICH THE OPERATION IS APPLIED MUST BELONG TO THE SAME TYPE
  - NORMALLY ILLEGAL TO ADD AN Integer VALUE TO A Float VALUE.

INSTRUCTOR NOTES

1. 1 IS INTEGER LITERAL; TYPE MISMATCH WITH OTHER OPERAND Float_1

2. Float_3 HAS NOT BEEN DECLARED

3. TYPE MISMATCH BETWEEN OPERANDS INT_2 and Float_1

4. TYPE MISMATCH BETWEEN TARGET INT_1 AND EXPRESSION Float_1 + Float_2

5 MINUTES.

IN CLASS EXERCISE

```
procedure Find_the_Errors is
   Float_1, Float_2 : Float   := 0.0;      }  DECLARATIVE PART
   Int_1, Int_2     : Integer := 1;        }
begin -- Find_the_Errors
   Float_1 := Float_1 + 1;                 }
   Float_3 := Float_1 + Float_2;           }  EXECUTABLE PART
   Int_1   := Float_1 + Int_2;             }
   Int_1   := Float_1 + Float_2;           }
end Find_the_Errors;
```

INSTRUCTOR NOTES

IF STUDENTS ARE DISTRESSED BY UNFAMILIAR MATERIAL IN THIS EXAMPLE, BRIEFLY EXPLAIN:

- FUNCTION SPEC
  - HAS AN ARGUMENT AND ARGUMENT TYPE
  - RETURN TYPE SPECIFIED
- "with" AND "use" TELL THE COMPILER TO IMPORT Gamma FUNCTION
- Stirling_Factorials *AND* Gamma *FUNCTIONS* WERE PLACED ON THIS EARTH BY A THOUGHTFUL PROVIDENCE TO BE USED AS HIGHFALUTIN' EXAMPLES IN COURSES; OTHERWISE THEY ARE OF NO USE TO ANYONE*

POINT OUT THAT IF Math_Pac HAD NOT BEEN COMPILED FIRST, THE COMPILER WOULD COMPLAIN ABOUT AN UNDECLARED IDENTIFIER.

*"THAT'S THE WAY IT IS!" -- NEVER MIND THAT THAT'S NOT PRECISELY TRUE.

STRONG TYPING

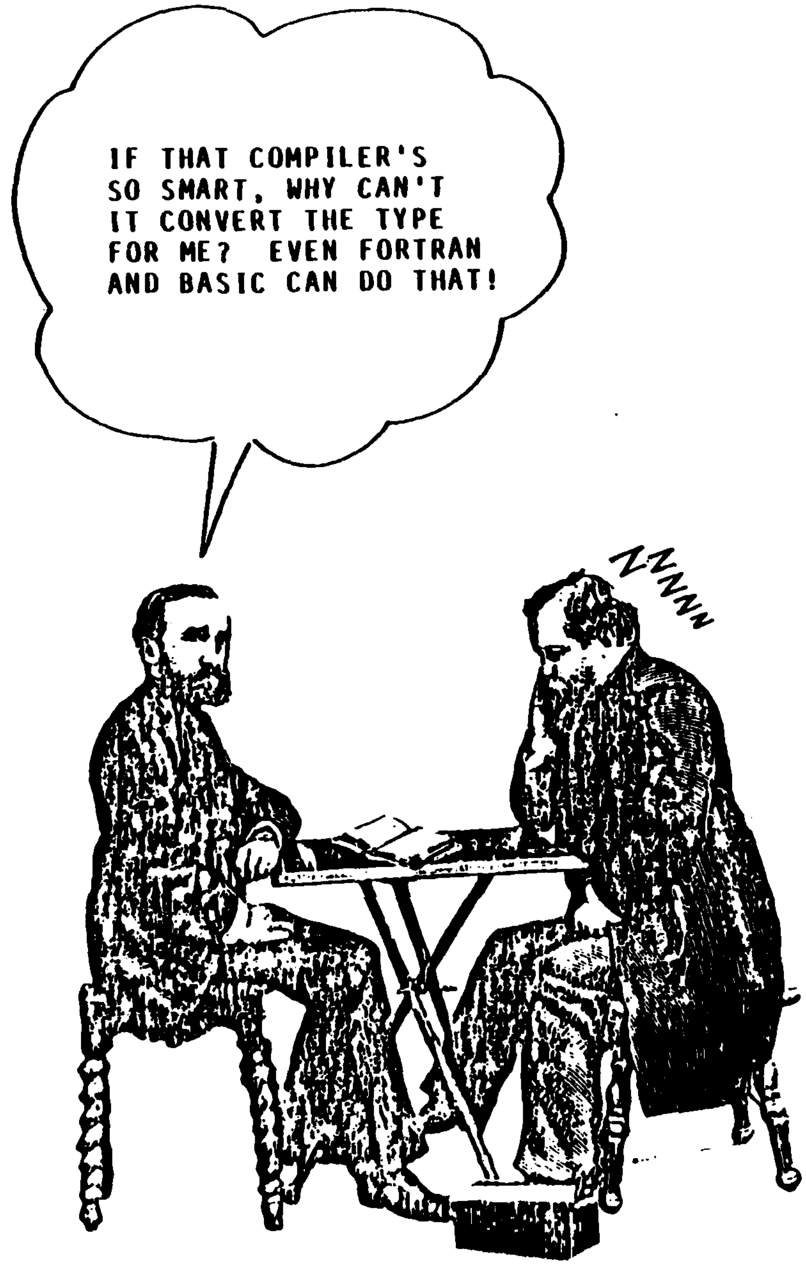
IN Ada, STRONG TYPING IS ENFORCED THROUGHOUT, EVEN BETWEEN SEPARATELY-COMPILED PROGRAM UNITS:

```
package Math_Pac is
   function Gamma (X : Float) return Float;
      ...
end Math_Pac;
```

```
with Math_Pac; use Math_Pac;
procedure Stirling_Factorials is
   N : Integer;
   A : Float;
begin -- Stirling_Factorials
      ...
   A := Gamma (N);
      ...
end Stirling_Factorials;
```

EVEN THOUGH THESE TWO UNITS WERE COMPILED AT DIFFERENT TIMES, THE COMPILER WILL DETECT THAT THE ARGUMENT SUPPLIED DOES NOT HAVE THE CORRECT TYPE (AND THE CODE IS THEREFORE INVALID).

INSTRUCTOR NOTES


IF THAT COMPILER'S SO SMART, WHY CAN'T IT CONVERT THE TYPE FOR ME? EVEN FORTRAN AND BASIC CAN DO THAT!
ZZNNN

INSTRUCTOR NOTES

TYPE CONVERSION

ONE COULD SAY THAT THE COMPILER IS SMART ENOUGH NOT TO CONVERT TYPES AUTOMATICALLY; EXPERIENCE INDICATES THAT WHAT LOOKS LIKE A SITUATION CALLING FOR AN IMPLICIT CONVERSION IS VERY OFTEN A PROGRAMMING ERROR.

ALSO, IMPLICIT CONVERSIONS INTRODUCE EXTRA CODE, UNBEKNOWNST TO THE PROGRAMMER, WHICH CAN DEGRADE EFFICIENCY.

INSTRUCTOR NOTES

POINT OUT THAT THEY WORK LIKE FUNCTIONS AND RETURN A VALUE OF THE SPECIFIED TYPE.

CONVERSION IS ALLOWED BETWEEN NUMERIC TYPES (E.G., Integer AND Float), AND IN CERTAIN OTHER CIRCUMSTANCES. CONVERSION OF A Float VALUE TO TYPE Integer INVOLVES ROUNDING (NOT TRUNCATION! CONVERSION FOR .5 IS IMPLEMENTATION DEPENDENT, E.G. WHETHER 3.5 BECOMES 3 OR 4 IS NOT DEFINED BY THE LANGUAGE.)

CONVERSION IS NOT ALLOWED BETWEEN Integer VALUES AND CHARACTERS, BUT THERE ARE OTHER MECHANISMS WE'LL GET TO LATER TO TRANSLATE BETWEEN CHARACTERS AND THEIR ASCII CODES.

TYPE CONVERSION

IF YOU WANT A TYPE CONVERSION, NO PROBLEM! JUST SAY SO!

SYNTAX:

```
Target_Object_Name := Type_Name (Expression);
```

EXAMPLE:

```
procedure An_Example is
        Int_1, Int_2     : Integer := 3;
        Float_1, Float_2 : Float   := 4.6;
begin -- An_Example
        Int_1   := Integer (Float_2);      -- Int_1 is now 5
        Float_1 := Float (Int_2);          -- Float_1 is now 3.0
end An_Example;
```

GENERAL RULE:

TO CONVERT ANY NUMERIC EXPRESSION TO ANY DESIRED NUMERIC TYPE, WRITE THE TYPE NAME FOLLOWED BY THE EXPRESSION TO BE CONVERTED ENCLOSED IN PARENTHESES. MAKE SURE THE CONVERSION "MAKES SENSE."

INSTRUCTOR NOTES

POINT OUT THAT IN THE LAST EXAMPLE

```
Int_1 := Integer (Float_1 + Float_2);
```

Float_1 AND Float_2 ARE ADDED AS FLOATING POINT NUMBERS AND THEN CONVERTED TO Integer.

IT WOULD ALSO HAVE BEEN POSSIBLE TO WRITE Int_1 := Integer (Float_1) + Integer (Float_2), POSSIBLY WITH DIFFERENT RESULTS. (IF Float_1 AND Float_2 BOTH HOLD 0.6, Integer (Float_1 + Float_2) = Integer (1.2) = 1, BUT Integer (Float_1) + Integer (Float_2) = 1 + 1 = 2.)

SOMEONE MAY ASK ABOUT CONVERTING 3.5 TO Integer AND WHETHER IT IS ROUNDED UP OR DOWN. THE ANSWER IS "IT IS IMPLEMENTATION DEPENDENT." THE IMPLEMENTATION MAY DO EITHER.

3.4 ROUNDS TO 3.

3.6 ROUNDS TO 4.

ASSIGN EXERCISE 3.

TYPE CONVERSION

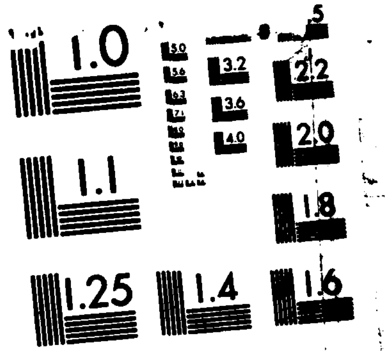
EXAMPLE:

```
procedure No_More_Errors is
   Float_1, Float_2 : Float   := 0.0;
   Int_1, Int_2 : Integer := 1;
   Float_3 : Float;
begin -- No_More_Errors
   Float_1 := Float_1 + 1.0;
   Float_3 := Float_1 + Float_2;         -- this is okay as is,
                                         -- now that we've given Float_3 a type
   Int_1 := Integer (Float_1) + Int_2;   -- now the operands agree
   Int_1 := Integer (Float_1 + Float_2); -- now the type of the assigned
                                         -- expression agrees
                                         -- with the target variable
end No_More_Errors;
```

INSTRUCTOR NOTES

AD-A166 366 ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA PROGRAMMING L202 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 4/8

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
3.2
3.6
4.0

TYPE CONVERSION

```
package Math_Pac is
   function Gamma (X : Float) return Float;
      ...
end Math_Pac;
```

```
with Math_Pac; use Math_Pac;
procedure Stirling_Factorials is
   N : Integer;
   A : Float;
begin -- Stirling_Factorials
      ...
   A := Gamma (Float (N));
      ...
end Stirling_Factorials;
```

THE COMPILER SMILES.

INSTRUCTOR NOTES

1. "YES, VIRGINIA, OBJECTS REALLY DO CEASE TO EXIST. THEIR NAMES CAN BE REDEFINED, WHATEVER MEMORY THEY OCCUPIED IS AGAIN AVAILABLE, AND SO ON."

2. "YES, THE DECLARATIVE PART CAN PRODUCE EXECUTABLE CODE -- THAT, HOWEVER, IS AN IMPLEMENTATION ISSUE WITH WHICH THE Ada PROGRAMMER IS NOT NORMALLY CONCERNED."

EXAMPLE:

```
I : constant Integer := 2 * N;
```

N IS A VARIABLE DECLARED OUTSIDE OF THIS PROCEDURE.

I IS A CONSTANT THROUGHOUT THE EXECUTION OF THIS PROCEDURE, AND THEN CEASES TO EXIST. AT THE NEXT EXECUTION, I WILL AGAIN BE CONSTANT, BUT WITH A DIFFERENT VALUE.

POINT OUT THAT Count AND Pi ARE "LOCAL" TO THE PROCEDURE.

ELABORATION

- "PROCESS BY WHICH A DECLARATION ACHIEVES ITS EFFECT"

  -- Ada LANGUAGE REFERENCE MANUAL

- MAKES THE OBJECT BEING DECLARED COME INTO EXISTENCE AND ASSIGNS ANY INITIAL VALUE TO IT

- DECLARATIONS ARE ELABORATED IN ORDER AND MAY REFER TO PREVIOUS DECLARATIONS

  EXAMPLE:

```
procedure P is
      Count  :  Integer;                       -- THESE DECLARATIONS ARE MADE
      Pi     :  constant Float:= 3.14159;      -- ANEW EVERY TIME PROCEDURE P
      Two_Pi :  constant Float:= 2.0 * Pi;     -- IS EXECUTED
begin --  P
      ...
end P;                                         -- EVERY OBJECT THAT WAS
                                               -- DECLARED
                                               -- WITHIN PROCEDURE P CEASES TO
                                               -- EXIST WHEN PROCEDURE P
                                               -- TERMINATES
```

INSTRUCTOR NOTES

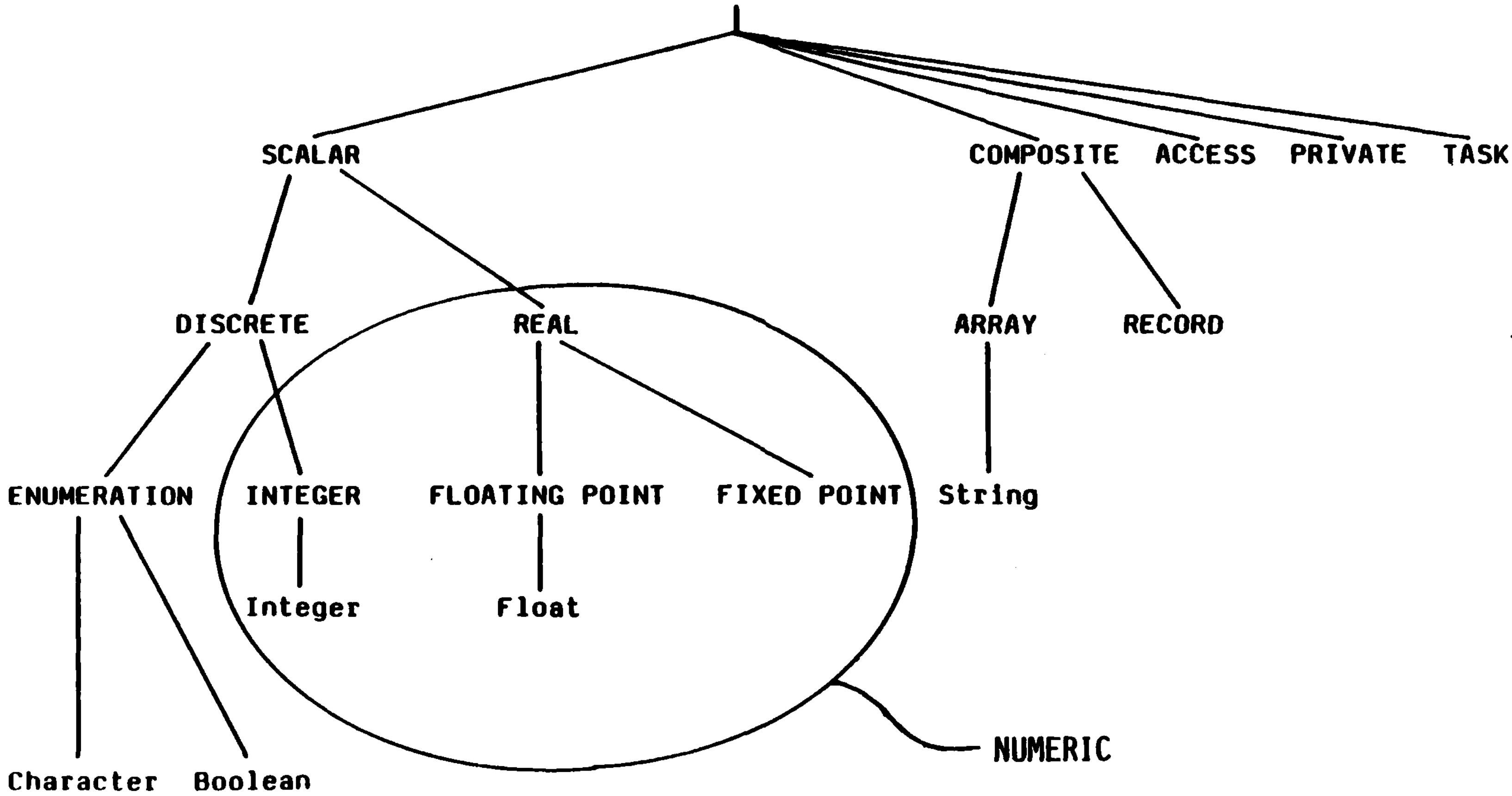
CLASSES OF TYPES ARE IN UPPER CASE, INDIVIDUAL TYPES ARE IN LOWER CASE. "INTEGER" IS THE NAME OF BOTH A CLASS OF TYPES AND AN INDIVIDUAL TYPE. THE CLASS OF INTEGER TYPES INCLUDES THE PREDEFINED TYPE Integer, OTHER PREDEFINED TYPES, AND USER-DEFINED TYPES.

POINT OUT THAT THIS COURSE DOES NOT COVER private OR task TYPES.

USER-DEFINED TYPES ARE NOT SHOWN. EVERY CLASS OF TYPES MAY INCLUDE USER-DEFINED TYPES.

private TYPES INCLUDE limited private.

A D A T Y P E S



VG 728.2

INSTRUCTOR NOTES

IF POSSIBLE DISPLAY PREVIOUS PAGE WHILE EXPLAINING THESE POINTS.

SCALAR TYPES

- A SCALAR IS AN ENTITY WHICH CANNOT BE DECOMPOSED ANY FURTHER

- SCALAR OBJECTS CAN ONLY ASSUME ONE VALUE AT A TIME

- SCALAR TYPES MAY BE EITHER "DISCRETE" OR "REAL"
    - DISCRETE SCALAR TYPES
        - Enumeration
        - Integer
    - REAL SCALAR TYPES
        - Floating
        - Fixed

- ALL SCALAR TYPES ARE ORDERED
    - EVERY VALUE IN THE TYPE IS EITHER LESS THAN OR GREATER THAN EVERY OTHER VALUE.
    - VALUES HAVE IMMEDIATE PREDECESSORS AND SUCCESSORS

INTEGER AND REAL SCALAR TYPES ARE COLLECTIVELY KNOWN AS NUMERIC TYPES

INSTRUCTOR NOTES

- THIS FOIL " WETS THE WHISTLE" FOR MORE TYPING TO FOLLOW AND PROVIDES MOTIVATION FOR NAMED NUMBERS, THE NEXT AND LAST SUBJECT OF THIS SECTION.

- Interrupt_Status_Type IS AN ENUMERATION TYPE WITH TWO VALUES, Yes AND No.

- Char_Count_Type IS AN INTEGER TYPE WITH 351 VALUES

- Volume_in_Liters_Type IS A FIXED POINT TYPE WITH AT LEAST 50,001 DISTINCT VALUES

- Bit_Sequence_Type IS EXACTLY WHAT IT APPEARS TO BE, AN 8-BIT BINARY NUMBER (BUT OF COURSE, NOT A NUMERIC VALUE THAT CAN USE NUMERIC OPERATIONS!)

QUESTION: WHAT IF I TRY TO ASSIGN A VARIABLE OF TYPE Char_Count_Type THE VALUE "-1"? (MAKE SURE THE STUDENT SAID A <u>VARIABLE</u> OR <u>OBJECT</u>. IF NOT, HE IS TRYING TO ASSIGN TO A <u>TYPE</u>, WHICH OF COURSE IS ILLEGAL!)

ANSWER: IT IS TREATED AS AN ERROR. WAIT UNTIL SECTION 10 FOR FURTHER DETAILS.

- TYPE DECLARATIONS ARE COVERED IN DETAIL IN THE NEXT SECTION

- TYPE DECLARATIONS ARE "BLUEPRINTS" WHICH TELL THE COMPILER HOW TO MAKE THE OBJECTS.

- THE TYPE DECLARATION (BLUEPRINT) MUST BE DEFINED BEFORE ANY OBJECTS OF THAT TYPE CAN BE DECLARED.

USER-DEFINED TYPES: TYPE DECLARATIONS

SYNTAX:

```
type Some_Type_Name is Type_Definition;
--   Some_Type_Name is referred to as the type name
```

EXAMPLES:

```
type Interrupt_Status_Type is (Yes, No);
type Char_Count_Type is range 0 .. 350;
type Volume_In_Liters_Type is delta 0.001 range 0.0 .. 50.0;
type Bit_Sequence_Type is array (0 .. 7) of Boolean;

Interrupt_Status:  Interrupt_Status_Type := Yes;
```

INSTRUCTOR NOTES

QUESTION: WHY IS IT BETTER PROGRAMMING PRACTICE ...?

ANSWER: THE COMPILER IS MEANT TO WORK FOR YOU, NOT VICE VERSA. IF YOU SPECIFY THE TYPE, THEN YOU ARE DENYING THE COMPILER THE OPPORTUNITY TO CHOOSE THE BEST INTERNAL REPRESENTATION. FURTHERMORE, YOU MAY UNNECESSARILY RESTRICT THE SITUATIONS IN WHICH YOU CAN USE THIS CONSTANT BECAUSE OF Ada'S STRONG TYPING RULES. STRONG TYPING IS DISCUSSED LATER.

NAMED NUMBERS

- SYMBOLIC NAMES FOR NUMERIC VALUES.

- NO TYPE IS SPECIFIED

- DECLARED IN NUMBER DECLARATIONS:

SYNTAX:

```
Name {, Name} :  constant := Numeric_Value;
```

- DIFFERENCES BETWEEN NUMBER DECLARATIONS AND CONSTANT DECLARATIONS:
  - -- NO TYPE NAME FOLLOWING THE WORD CONSTANT IN NUMBER DECLARATIONS
  - -- INITIAL VALUE MUST BE NUMERIC, WITHOUT ANY TYPE SPECIFIED.

- IN GENERAL, IT IS BETTER PROGRAMMING PRACTICE TO USE A NAMED NUMBER RATHER THAN A NUMERIC CONSTANT.

INSTRUCTOR NOTES

"NOTE THAT IN THIS EXAMPLE THE NAMED NUMBER Pi IS USED AS TWO DISTINCT VALUES, ONE OF TYPE Float AND ONE OF TYPE Attitude_Type. (THIS IS ALSO THE CASE WITH THE LITERAL 2.0, INCIDENTALLY).

HAD WE EXPLICITLY DECLARED A TYPE FOR THE CONSTANT Pi, ONLY ONE OF THE LAST TWO STATEMENTS WOULD HAVE BEEN LEGAL."

FOR EXAMPLE:

```
Pi : constant Float := 3.14159;
```

WOULD MAKE THE if STATEMENT ILLEGAL.

NAMED NUMBER EXAMPLE

CONTEXT:

```
Pi : constant := 3.14159;
type Attitude_Type is digits 5 range -180.0 .. 180.0;
Pitch, Roll, Yaw : Attitude_Type;
Circumference    : Float;
Radius           : Float;
```

EXAMPLES:

```
Circumference := Radius * 2.0 * Pi;
                 \________________/
                       Float

if Roll < Pi/2.0 then ...
   \__/   \____/
  Attitude_Type
```

- <u>POINT</u>: NAMED NUMBERS CAN BE USED IN EXPRESSIONS OF ANY TYPE WITHIN THEIR CLASS.

INSTRUCTOR NOTES

POINT OUT THAT THIS IS JUST AN INTRODUCTION TO ATTRIBUTES. EACH TYPE HAS ATTRIBUTES APPLICABLE TO THE TYPE AND WE WILL ADDRESS THE ATTRIBUTES FOR A SPECIFIC TYPE WHEN WE GET TO THAT TYPE.

ATTRIBUTES

- Ada CONTAINS NUMEROUS PREDEFINED ATTRIBUTES* WHICH CONTAIN INFORMATION ABOUT DATA TYPES.

- DENOTED BY APOSTROPHE FOLLOWED BY ATTRIBUTE

```
Type_Name'Attribute_Identifier [(parameter)]
```

*LISTED IN APPENDIX A OF LRM

INSTRUCTOR NOTES

POINT OUT THAT IF YOU HAVE YOUR OWN INTEGER TYPE

```
type My_Int is range 50 .. 100;
```

My_Int'First IS 50. DO THIS ONLY IF YOU STRESSED USER DEFINED SLIDE PREVIOUSLY.

THE RATIONALE FOR THIS IS MAINTAINABILITY. IT WILL NOT CHANGE FROM IMPLEMENTATION TO IMPLEMENTATION.

ATTRIBUTES OF INTEGER TYPES

- T'First - IDENTIFIES SMALLEST VALUE OF THE INTEGER TYPE T

- T'Last - IDENTIFIES LARGEST VALUE OF THE INTEGER TYPE T

NOTE: Integer'First AND *Integer*'Last ARE MACHINE-DEPENDENT

- ON A 16-BIT MACHINE WITH TWO'S COMPLEMENT ARITHMETIC

  Integer'First YIELDS -32768

  Integer'Last YIELDS +32767

INSTRUCTOR NOTES

POINT OUT THAT THIS IS REQUIRED FOR I/O FOR OBJECTS OF PREDEFINED TYPE Integer AND Float.

POINT OUT THAT with AND use REFER BACK TO SECTION 2, Ada TECHNICAL OVERVIEW.

I/O

STEP 1: AT TOP OF COMPILATION UNIT WRITE

```
with Text_IO; use Text_IO;
```

STEP 2: IN DECLARATIVE PART WRITE

```
--
--  for Integer I/O
--
package Int_IO is new Integer_IO (Integer);
use Int_IO;

-- or

--
-- for Float I/O
--
package Flt_IO is new Float_IO (Float);
use Flt_IO;
```

INSTRUCTOR NOTES

POINT OUT:

1. OBJECT DECLARATIONS IN DECLARATIVE PART

2. MULTIPLE OBJECT DECLARATIONS/LINE

3. ACCESS TO IO VIA with AND use CLAUSE AS SPECIFIED IN STEP 1.

4. INSTANTIATION AS IN STEP 2.

5. THAT YOU COULD HAVE DECLARED A THIRD INTEGER OBJECT, SUM : INTEGER; AND IN THE EXECUTABLE PORTION WRITTEN

```
Sum := Value_1 + Value_2;
Put (Sum);
```

ASSIGN EXERCISES 3 AND 4 NOW.

ASSIGN CHAPTER 4 OF THE PRIMER.

AN EXAMPLE

```
with Text_IO; use Text_IO;
procedure Do_Sum is

        --
        -- for the predefined type Integer
        --
        Value_1, Value_2 : Integer;

        package Int_IO is new Integer_IO (Integer);
        use Int_IO;

begin -- Do_Sum

        Get (Value_1);
        Get (Value_2);
        Put (Value_1 + Value_2);

end Do_Sum;
```

INSTRUCTOR NOTES

ALLOCATE TWO (2) HOURS FOR THIS SECTION. ASSIGN EXERCISES 5, 6, 7 AND 8 AT THE COMPLETION OF THIS SECTION.

THE OBJECTIVE OF THIS SECTION IS TO INTRODUCE THE CONCEPT OF ENUMERATION TYPES, OPERATIONS FOR ENUMERATION TYPES, AND THE if AND case CONTROL STRUCTURES.

SECTION 5
ENUMERATION TYPES AND CONTROL STRUCTURES

INSTRUCTOR NOTES

A DICTIONARY DEFINITION.

"THE BIGGEST CAUSE OF CONFUSION ABOUT ENUMERATION TYPES HAS BEEN IN THE PAST, WHAT IT MEANS."

DO NOT SPEND A LOT OF TIME ON THIS SLIDE.

ENUMERATION TYPES

enumerate (i-n(y)u-me-rat) v.t.

1. to ascertain the number of; to count

2. to specify one after another; to list

INSTRUCTOR NOTES

"ENUMERATION TYPE DECLARATIONS LIST, I.E. SPECIFY ONE AFTER ANOTHER, THE POSSIBLE VALUES FOR OBJECTS OF THAT TYPE."

ENUMERATION TYPE VALUES

- EVERY TYPE HAS A SET OF VALUES AND A SET OF OPERATIONS.
- THE SET OF VALUES OF EACH PREDEFINED TYPE, FOR EXAMPLE, PREDEFINED Integer, IS DEFINED BY THE SYSTEM.
- AN ENUMERATION TYPE IS ONE WHERE THE PROGRAMMER DEFINES BY LISTING, IN ORDER, ALL THE VALUES THAT VARIABLES AND CONSTANTS (I.E. OBJECTS OF THE TYPE) ARE PERMITTED TO ASSUME.

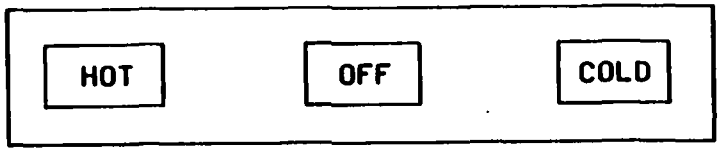
INSTRUCTOR NOTES

POINT OUT THAT NUMERIC VALUES ARE BEING USED. THIS WILL BE DIFFICULT FOR A MAINTAINER TO RECALL. A MAINTAINER NEEDS TO BE CONCERNED WITH MAKING CHANGES WITHOUT IMPACTING OTHER CODE, NOT WITH REMEMBERING WHETHER HOT IS -1 OR 1.

POINT OUT THAT ASSIGNING 5 TO Control_Panel_Switch IN OTHER LANGUAGES WOULD PASS THROUGH THE COMPILER UNDETECTED. BUT THE ASSIGNMENT IS MEANINGLESS AS THE ONLY MEANINGFUL VALUES ARE 1, 0, AND -1. BY USING ENUMERATION TYPE DECLARATIONS, ONLY HOT, OFF, AND COLD ARE LEGAL VALUES. ANY ATTEMPT TO ASSIGN A VALUE OTHER THAN HOT, OFF, OR COLD WILL BE DETECTED AT COMPILE TIME.

SIMILARLY THE CONDITION CONTROL_PANEL_SWITCH = 3 IS MEANINGLESS AS THERE IS NO TOGGLE REPRESENTED BY 3.

WHY ENUMERATION TYPES?



CONTROL PANEL

IN SOME LANGUAGES

```
HOT  = -1
OFF  =  0
COLD =  1
```

DISADVANTAGE:

SUPPOSE THE INTEGER VARIABLE Control_Panel_Switch REPRESENTS THE ABOVE PANEL. THE FOLLOWING CODE IS LEGAL BUT ILLOGICAL:

```
•   Control_Panel_Switch := 5;         -- who will catch the error?
•   if Control_Panel_Switch = 3 then   -- what does it mean?
       ...
    end if;
```

INSTRUCTOR NOTES

POINT OUT USE OF WORD "TYPE" IN NAME OF THE TYPE.

POINT OUT USE OF THE TYPE NAME <u>WITHOUT</u> THE WORD TYPE FOR USE AS AN IDENTIFIER NAME.

STRESS THAT THIS IS JUST A QUICK LOOK TO SEE WHAT AN ENUMERATION TYPE AND OBJECT DECLARATION LOOKS LIKE IN Ada.

POINT OUT THAT THE <u>ONLY</u> ALLOWED VALUES FOR Control_Panel_Switch ARE Hot, Off, AND Cold, AND THAT Control_Panel_Switch IS A VARIABLE, LIKE X, AND CAN BE ASSIGNED TO, COMPARED, ETC.

POINT OUT THAT "HOT", "OFF", AND "COLD" ARE NAMES OF VALUES JUST AS 1 AND 2 ARE NAMES OF INTEGER VALUES.

ENUMERATION TYPES

IN Ada

```
-- THE TYPE DECLARATION
type Control_Panel_Switch_Type is (Hot, Off, Cold);

-- THE OBJECT DECLARATION
Control_Panel_Switch : Control_Panel_Switch_Type;
```

- THE OBJECT Control_Panel_Switch CAN HOLD ANY VALUE LISTED IN THE DECLARATION OF Control_Panel_Switch_Type : Hot, Off, OR Cold

INSTRUCTOR NOTES

POINT OUT THAT THE VERSION THAT USES ENUMERATION TYPES IS MORE READABLE.

POINT OUT THAT THE ... IS PROBABLY PROCEDURE CALLS.

ENUMERATION TYPES

- WHICH IS EASIER TO UNDERSTAND AND MAINTAIN?

CONTEXT:

```
Control_Panel_Switch : Integer;
```

EXAMPLE:

```
if Control_Panel_Switch = -1 then
   ...
elsif Control_Panel_Switch = 0 then
   ...
else
   ...
end if;
```

OR

CONTEXT:

```
type Control_Panel_Switch_Type is (Hot, Off, Cold);
Control_Panel_Switch : Control_Panel_Switch_Type;
```

EXAMPLE:

```
if Control_Panel_Switch = Hot then
   ...
elsif Control_Panel_Switch = Off then
   ...
else
   ...
end if;
```

INSTRUCTOR NOTES

STRESS THAT ALL THE VALUES ARE LISTED IN THE ENUMERATION TYPE DECLARATION. POINT OUT ALSO THAT THEY ARE LISTED IN ORDER.

POINT OUT THAT, AS IN ALL TYPE DECLARATIONS, A SEMICOLON IS REQUIRED AT THE END OF AN ENUMERATION TYPE DECLARATION.

BULLET 1:

ASK THE CLASS WHICH IDENTIFIERS WOULD <u>NOT</u> BE LEGAL FOR Type_Name

(ANSWER: ANY RESERVED WORDS)

ENUMERATION TYPE DECLARATION

SYNTAX:

```
type Type_Name is (value_1, value_2, ... value_n);
```

where

- Type_Name IS ANY LEGAL IDENTIFIER
- value_i IS EITHER AN IDENTIFIER OR A CHARACTER LITERAL
- THE LIST OF ALLOWED VALUES FOR THE TYPE IS ENCLOSED IN PARENTHESES AND HAS AN IMPLIED ORDER

THE SPECIFIED VALUES ARE CALLED ENUMERATION LITERALS

INSTRUCTOR NOTES

POINT OUT THAT

```
type Baud_Rate_Type is (300, 600, 1200);
```

IS ILLEGAL BECAUSE ENUMERATION LITERALS ARE <u>IDENTIFIERS</u> (WHICH MUST BEGIN WITH A LETTER) OR CHARACTER LITERALS.

IN Validity_Type, RI STANDS FOR Routing Indicator AND LMF STANDS FOR Line Media Format

ENUMERATION TYPE DECLARATIONS

```
type File_Privilege_Type is (Read, Write, Edit, Delete);

type Character_Set_Type is (ASCII, ITA, EBCDIC);

type Day_Type is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);

type Baud_Rate_Type is (B300, B600, B1200);

type Counting_Format_Type is (Binary, BCD);

type Precedence_Type is (Routine, Priority, Immediate,
                         Flash, ECP, Critical);

type Validity_Type is (Valid, Bad_RI, Bad_LMF, Security_Mismatch);
```

NOTE:

```
type Baud_Rate_Type is (300, 600, 1200);  -- ** ILLEGAL
```

INSTRUCTOR NOTES

POINT OUT THAT EACH TYPE DEFINES A SET OF VALUES AND A SET OF OPERATIONS. HERE WE ARE ADDRESSING VALUES ONLY.

NOTE: Pvt IS USED HERE, SINCE private IS A RESERVED WORD.

ENUMERATION TYPE VALUES

```
type Army_Rank_Type is (Pvt, Sergeant, Lieutenant, Captain);
```

- AT ANY MOMENT, AN OBJECT OF TYPE Army_Rank_Type MAY HOLD ONE OF THE FOLLOWING VALUES:

  Pvt

  Sergeant

  Lieutenant

  Captain

```
type Baud_Rate_Type is (B300, B600, B1200);
```

- AT ANY MOMENT, AN OBJECT OF TYPE Baud_Rate_Type MAY HOLD ONE OF THE FOLLOWING VALUES:

  B300

  B600

  B1200

OBJECTS IN THESE TYPES MAY NOT HOLD ANY VALUES OTHER THAN THOSE LISTED ABOVE.

INSTRUCTOR NOTES

YOU WANT TO POINT OUT THAT THE INTERNAL REPRESENTATION OF THE ENUMERATION LITERAL IS USUALLY CHOSEN BY THE COMPILER. THE PROGRAMMER COULD EXPLICITLY SPECIFY THEM WITH REPRESENTATION SPECIFICATIONS. THIS SHOULD ONLY BE DONE WITH GOOD REASON BECAUSE IT COULD MAKE THE CODE MUCH LESS EFFICIENT.

POINT OUT THAT SINCE VALUES ARE ORDERED, THE RELATIONAL OPERATIONS ARE APPLICABLE.

COMMENTS ON ENUMERATION TYPE VALUES

- VALUES OF AN ENUMERATION TYPE ARE CALLED ENUMERATION LITERALS.

- USER DOES NOT NEED TO BE AWARE OF INTERNAL REPRESENTATION OF ENUMERATION LITERALS.

- ENUMERATION VALUES MUST BE IDENTIFIERS OR CHARACTER LITERALS

- VALUES DEFINED IN AN ENUMERATION TYPE FORM AN ORDERED SET AND ARE ORDERED BY POSITION IN THE DECLARATION.

  - EXAMPLE:
    ```
    type Army_Rank_Type is (Pvt, Sergeant, Lieutenant,
                            Captain);
    ```

  - NOTICE:
    ```
    Sergeant < Lieutenant
    Sergeant > Pvt
    ```

INSTRUCTOR NOTES

POINT OUT TO THE STUDENT THAT IT IS THE OBJECT DECLARATION WHICH ALLOCATES STORAGE.

OBJECT DECLARATIONS HERE ARE NO DIFFERENT FROM NUMERIC OBJECT DECLARATIONS.

DECLARATION OF OBJECTS

TYPE DECLARATION: type File_Privilege_Type is (Read, Write, Edit, Delete);

OBJECT DECLARATION: Group_Privilege, User_Privilege : File_Privilege_Type;

Group_Privilege AND User_Privilege MAY HAVE VALUES OF Read, Write, Edit, OR Delete

TYPE DECLARATION: type Baud_Rate_Type is (B300, B600, B1200);

OBJECT DECLARATION: Baud_Rate : Baud_Rate_Type := B1200;

Baud_Rate MAY HAVE VALUES OF B300, B600, OR B1200, AND ITS INITIAL VALUE IS B1200

INSTRUCTOR NOTES

ASSIGNING AN ENUMERATION VALUE TO AN ENUMERATION OBJECT IS THE SAME AS IN NUMERIC TYPES.

ENUMERATION TYPE OPERATIONS

ASSIGNMENT

:=

CONTEXT	EXAMPLES
type Baud_Rate_Type is (B300, B600, B1200); Baud_Rate : Baud_Rate_Type;	Baud_Rate := B600;
type Counting_Format_Type is (Binary, BCD); Counting_Format : Counting_Format_Type;	Counting_Format := Binary;

INSTRUCTOR NOTES

HERE WE ARE TALKING ABOUT OPERATIONS AVAILABLE ON ENUMERATION TYPES.

"RELATIONAL EXPRESSIONS HAVE THE VALUE True OR False, AND ARE TYPICALLY USED IN CONTROL STRUCTURES SUCH AS if STATEMENTS."

ARITHMETIC OPERATIONS ARE NOT ALLOWED. IT DOESN'T MAKE SENSE TO ADD VALUES OF TYPE Precedence_Type. ADDING Routine AND Flash IS MEANINGLESS.

ENUMERATION TYPE OPERATIONS

RELATIONAL

= /= < <= > >=

CONTEXT	EXAMPLES

type Validity_Type is (Valid, Bad_RI,
Bad_LMF, Security_Mismatch);
Status : Validity_Type;
	if Status = Valid then
	.
	.
	.
	end if;

type Precedence_Type is (Routine, Priority, Immediate, Flash, ECP, Critical); Message_Level : Precedence_Type;	while Message_Level /= Critical loop . . . end loop;

type Baud_Rate_Type is (B300, B600, B1200);
Baud_Rate : Baud_Rate_Type;
	if Baud_Rate < B1200 then
	.
	.
	.
	end if;

- REMEMBER THAT ENUMERATION VALUES HAVE AN IMPLIED ORDER DEFINED BY THEIR APPEARANCE IN THE TYPE DECLARATION

INSTRUCTOR NOTES

First IS AN ATTRIBUTE WHICH RETURNS THE FIRST VALUE.

Pred IS AN ATTRIBUTE WHICH RETURNS THE PREDECESSOR OF A VALUE. Pred WORKS ON DISCRETE TYPES WHICH ARE ORDERED.

SOME ATTRIBUTES OPTIONALLY TAKE PARAMETERS.

ATTRIBUTES OF DATA TYPES

Ada CONTAINS NUMEROUS PREDEFINED ATTRIBUTES WHICH CONTAIN INFORMATION ABOUT DATA TYPES.

AN ATTRIBUTE IS DENOTED BY AN APOSTROPHE FOLLOWED BY A SPECIAL IDENTIFIER:

```
Type_Name'Attribute_Identifier [(parameter)]
```

CONTEXT:

```
type Movie_Rating_Type is (G_Rated, PG_Rated, PG_13_Rated,
                           R_Rated, X_Rated);
```

EXAMPLE:

```
Movie_Rating_Type'First            -- YIELDS G_Rated
Movie_Rating_Type'Pred(X_Rated)    -- YIELDS R_Rated
```

INSTRUCTOR NOTES

FOR EXAMPLE Movie_Rating_Type'Last IS X_Rated AND Movie_Rating_Type'First IS G_Rated.

ATTRIBUTES APPEAR AS PARTS OF EXPRESSIONS; THEREFORE, THERE IS NO SEMI-COLON.

Val TAKES AN INTEGER VALUE AND RETURNS A VALUE, USING THE INTEGER VALUE AS AN INDEX INTO THE SET OF VALUES DEFINED BY THE TYPE DEFINITION. POINT OUT THAT THE INDEX BEGINS AT ZERO, NOT ONE. THUS E'Pos (E'First) = 0.

FOR THE INSTRUCTOR'S INFORMATION ONLY: THE ATTRIBUTES Pos AND Val OPERATE ON THE BASE TYPE OF THE TYPE IN QUESTION. THUS GIVEN A SUBTYPE OF Day_Type, Weekday_Type, WHOSE RANGE IS Mon .. Fri, Weekday_Type'Pos(Mon) = 1, NOT 0 AND Weekday_Type'Val(6) = Sat AND COULD POTENTIALLY RAISE Constraint_Error UPON ASSIGNMENT TO AN OBJECT OF Weekday_Type. THE REASON FOR THIS IS TO MAINTAIN THE INVERSE RELATIONSHIP BETWEEN Pos AND Val. (OTHERWISE, T'Val(2) COULD NOT RETURN A UNIQUE CONSISTENT RESULT.)

ENUMERATION TYPE ATTRIBUTES

GIVEN AN ENUMERATION TYPE E:

ATTRIBUTE	RETURNS
• E'First	THE FIRST ENUMERATION VALUE
• E'Last	THE LAST ENUMERATION VALUE
• E'Succ (Value)	THE SUCCESSOR OF Value
• E'Pred (Value)	THE PREDECESSOR OF Value
• E'Pos (Value)	THE POSITION IN THE LIST OF Value
• E'Val (Integer_Value)	THE ENUMERATION VALUE AT THE GIVEN POSITION

ANSWERS:

1. Wine_Type'First = Burgundy
2. Wine_Type'Last = Riesling
3. Wine_Type'Succ(Burgundy) = Chablis
4. Wine_Type'Pred(Pinot_Noir) = Chenin_Blanc
5. Wine_Type'Pos(Retsina) = 5
6. Wine_Type'Val(2) = Chablis

- COMMENTS ON RUNTIME ERROR NOTE:
  - ATTRIBUTES DO NOT WORK IN A CIRCULAR FASHION.

    Wine_Type'Succ (Riesling) IS UNDEFINED, NOT Burgundy.

  - Wine_Type'Val(7) RESULTS IN AN ERROR BECAUSE Wine_Type'Pos(Riesling) is 6.

    (F.Y.I. : Retsina is a Greek wine.)

ATTRIBUTE EXERCISES

CONTEXT FOR EXERCISES:

```
type Wine_Type is  (Burgundy, Chablis, Chardonnay, Chenin_Blanc, Pinot_Noir, Retsina,
                    Riesling);
```

EXERCISES:

1. Wine_Type'First = ____________
2. Wine_Type'Last = ____________
3. Wine_Type'Succ(Burgundy) = ____________
4. Wine_Type'Pred(Pinot_Noir) = ____________
5. Wine_Type'Pos(Retsina) = ____________
6. Wine_Type'Val(2) = ____________

NOTE: EACH OF THE FOLLOWING WOULD RESULT IN A RUNTIME ERROR:

```
Wine_Type'Succ(Riesling)
Wine_Type'Pred(Burgundy)
Wine_Type'Val(7)
```

INSTRUCTOR NOTES

THE DEFINITION OF type Boolean IS

```
type Boolean is (False, True);
```

MAKE SURE STUDENTS UNDERSTAND THAT THIS IS PREDEFINED AND SHOULD NOT APPEAR IN PROGRAMS. IT <u>IS</u> LEGAL (BUT STUPID) TO INCLUDE THIS DEFINITION IN A PROGRAM. DON'T BELABOR THE POINT.

POINT OUT THAT BOOLEAN OBJECTS ARE ASSIGNED VALUES OF BOOLEAN EXPRESSIONS.

BOOLEAN

- type Boolean IS A PREDEFINED ENUMERATION TYPE
- AN OBJECT OF type Boolean CAN HAVE A VALUE OF False OR True
- Boolean EXPRESSIONS YIELD A Boolean RESULT (True OR False) WHEN EVALUATED

CONTEXT:

```
Year      : Integer;
Leap_Year : Boolean;
```

EXAMPLE:

```
Leap_Year := (Year mod 4 = 0 and Year mod 100 /= 0) or (Year mod 400 = 0);
```

INSTRUCTOR NOTES

if STATEMENTS ARE DISCUSSED NEXT.

ITERATIVE CONTROL STRUCTURES, OR LOOPS, ARE DISCUSSED IN SECTION 8.

BOOLEAN EXPRESSIONS

- MOST OFTEN FOUND AS CONDITIONS IN if STATEMENTS AND ITERATIVE CONTROL STRUCTURES

CONTEXT:

```
type Person_Type is (Self, Other);
Doer       : Person_Type;
Competence : Integer range 0 .. 10;
type Desire_Type is (Job_Done_Right, Ignore_Job, Do_Adequate_Job);
Desire     : Desire_Type;
```

EXAMPLE:

```
if Competence > 0 and Desire = Job_Done_Right then
   Doer := Self;
else
   Pay_Through_The_Nose;
end if;
```

INSTRUCTOR NOTES

DRAW ATTENTION TO THE SHORT CIRCUIT CONTROL FORMS. THEY ARE EXPLAINED IN SECTION 7.

STUDENTS MAY BE UNFAMILIAR WITH THE LOGICAL OPERATOR "NOT". STRESS THAT THE FORM USED HERE IS PREFERABLE TO THE MORE FAMILIAR "if Equal = False then ..." (A CARRY-OVER FROM FORTRAN PROGRAMMING.

OPERATIONS YIELDING BOOLEAN RESULTS

CLASS	OPERATIONS	ARGUMENT TYPE
LOGICAL OPERATIONS	and or xor not	BOOLEAN
SHORT-CIRCUIT CONTROL FORMS	and then or else	BOOLEAN
EQUALITY TESTS	= /=	ANY
MEMBERSHIP TESTS	in not in	ANY
ORDERING TESTS	< <= > >=	ANY SCALAR TYPE

EXAMPLE OF A LOGICAL OPERATION:

CONTEXT:

```
Sensor_1_Enabled : Boolean;
```

EXAMPLE:

```
if not Sensor_1_Enabled then
   Enable_Sensor(Sensor_1);
end if;
```

INSTRUCTOR NOTES

THESE ARE THE SAME ATTRIBUTES AS FOR OTHER ENUMERATION TYPES.

POINT OUT THAT YOU COULD TAKE THE POSITION OF A Boolean VALUE AND USE IT IN AN EXPRESSION. FOR EXAMPLE,

```
Num_Days_In_February := 28 + Boolean'Pos (Leap_Year);
```

BOOLEAN TYPE ATTRIBUTES

- Boolean'First
- Boolean'Last
- Boolean'Succ (Boolean_Expression)
- Boolean'Pred (Boolean_Expression)
- Boolean'Pos (Boolean_Expression)
- Boolean'Val (Integer_Expression)

INSTRUCTOR NOTES

THIS PORTION ADDRESSES TWO OF Ada'S CONTROL STRUCTURES, THE if AND case STATEMENTS. EACH OF THESE CONSTRUCTS HAS MULTIPLE FORMS OF EXPRESSION. WE WILL EXAMINE ALL FORMS.

CONTROL STRUCTURES

- IF
  - ELSIF
  - ELSE
- CASE
  - BAR NOTATION
  - DISCRETE RANGE
  - "WHEN OTHERS"

INSTRUCTOR NOTES

POINT OUT THAT SPELLING OF elsif IS <u>NOT</u> "else if" BUT "elsif."

EXPLAIN THAT THE if STATEMENT IS A COMPOUND STATEMENT, WHICH MEANS THAT FROM THE WORD "if" TO THE "end if" IS CONSIDERED ONE STATEMENT.

THE if STATEMENT - TWO BASIC FORMS

FORM 1

```
if Condition then                      -- EVALUATE CONDITION (A Boolean EXPRESSION)
   statement(s);                       -- STATEMENT(S) EXECUTED ONLY IF Condition
                                       --       EVALUATES TO True
end if;                                -- CONTROL PASSES TO THE NEXT STATEMENT AFTER THE
                                       --       if STATEMENT
```

FORM 2

```
if Condition_1 then                    -- EVALUATE Condition_1
   statement(s);                       -- IF TRUE, STATEMENT(S) EXECUTED; CONTROL
                                       --       PASSES TO NEXT STATEMENT AFTER if
elsif Condition_2 then                 -- ELSE EVALUATE Condition_2
   statement(s);                       -- IF Condition_2 True, EXECUTE STATEMENTS; CONTROL
                                       --       PASSES TO NEXT STATEMENT
-- perhaps other elsif alternatives    -- ELSE CONTINUE EVALUATING Conditions
end if;                                -- IF NO Condition_i EVALUATES TO True, CONTROL
                                       --       PASSES TO NEXT STATEMENT AFTER THE if
                                       --       STATEMENT; NO ENCLOSED STATEMENTS
                                       --       ARE EXECUTED
```

INSTRUCTOR NOTES

THIS SLIDE AND THE NEXT TWO OFFER EXAMPLES OF if STATEMENTS. THIS IS AN EXAMPLE OF THE FIRST FORM.

POINT OUT THAT THE EXAMPLE CODE MIGHT BE PART OF A PROCEDURE TO INCREMENT THE CURRENT DATE, HANDLING THE SPECIAL CASE IN WHICH THE LAST DAY OF YEAR BECOMES THE FIRST DAY OF THE NEXT YEAR.

SOME EXAMPLES
FORM 1

CONTEXT:

```
type Month_Type is (January, February, March, April, May, June,
                    July, August, September, October, November,
                    December);
This_Month             :  Month_Type;
Day_Number, Year_Number:  Integer;
```

EXAMPLE:

```
if (This_Month = December) and (Day_Number = 31) then
      This_Month  := January;
      Day_Number  := 1;
      Year_Number := Year_Number + 1;
end if;
```

INSTRUCTOR NOTES

DON'T GET BOGGED DOWN IN QUESTIONS ON THE PARAMETER LIST. JUST INDICATE THAT in SPECIFIES THE FLOW OF DATA WITH RESPECT TO PROCEDURE Warn_Viewers.

THE POINT IS TO DISCUSS ENUMERATION TYPES WITH RESPECT TO THE if STATEMENT AND TO ADDRESS USE OF THE elsif TO PROVIDE ALTERNATIVE PATHS OF EXECUTION.

EXAMPLE FORM 2

THE PROCEDURE BELOW TAKES IN A MOVIE RATING AND PRINTS AN APPROPRIATE MESSAGE TO THE SCREEN FOR MOVIES RATED HIGHER THAN PG.

CONTEXT:

```
type Movie_Ratings_Type is (G_Rated, PG_Rated, PG_13_Rated,
                            R_Rated, X_Rated);
```

EXAMPLE:

```
procedure Warn_Viewers (Rating : in Movie_Ratings_Type) is

begin -- Warn_Viewers
   if Rating = X_Rated then
      Flash_On_Screen ("For Adults Only!");
   elsif Rating = R_Rated then
      Flash_On_Screen ("Under 17 Must Be Accompanied By Parent");
   elsif Rating = PG_13_Rated then
      Flash_On_Screen ("Parental Discretion Advised");
   end if;
end Warn_Viewers;
```

INSTRUCTOR NOTES

THESE ARE THE RULES FOR INCLUDING AN else ALTERNATIVE.

ELSE

- PROVIDES STATEMENTS TO BE EXECUTED IF ALL CONDITIONS EVALUATE TO FALSE
- MUST BE ONLY ONE else PER if STATEMENT
- IF APPEARING, IT MUST BE LAST

INSTRUCTOR NOTES

POINT OUT THAT THE ADDITION OF THE ELSE ALLOWS FOR STATEMENTS WITHIN THE if STATEMENT TO BE EXECUTED IF NONE OF THE CONDITIONS EVALUATE TO True.

BASIC IF STATEMENT FORMS WITH ELSE ALTERNATIVE

FORM 1 with else:

```
if Condition then
   statement(s);
else
   statement(s);
end if;
```

FORM 2 with else:

```
if Condition_1 then
   statement(s);
elsif Condition_2 then
   statement(s);
-- possibly other elsifs
else
   statement(s);
end if;
```

- STATEMENTS BETWEEN else AND end if ARE EXECUTED ONLY IF NO CONDITION EVALUATES TO True.

INSTRUCTOR NOTES

HERE'S AN else ALTERNATIVE USED WITH THE FIRST BASIC FORM.

EXAMPLE

CONTEXT:

```
type File_Privilege_Type is (Read, Write, Edit, Delete);
User_Privilege : File_Privilege_Type;
```

EXAMPLE:

```
if User_Privilege = Read then
   Get_Data;
else
   Put ("NO PRIVILEGE FOR READING FILE.");
end if;
```

INSTRUCTOR NOTES

POINT OUT THAT elsif ELIMINATES NESTED ifS. POINT OUT THAT end if IS TWO WORDS WHEREAS "elsif" IS ONE WORD.

THE STUDENTS MAY NOT BE CONVINCED BY THIS EXAMPLE. IF NECESSARY; PUT UP A GORY EXAMPLE WITH DIFFERENT DISCOUNT PERCENTS FOR 4 OR 5 LEVELS OF Gas_Pumped.

ANOTHER EXAMPLE

CONTEXT:

```
      Gas_Pumped, Discount_Percent : Float;
```

EXAMPLE:

```
      if Gas_Pumped > 40.0 then
             Discount_Percent := 10.0;
      elsif Gas_Pumped > 25.0 then
             Discount_Percent := 5.0;
      else
             Discount_Percent := 0.0;
      end if;
```

EQUIVALENT TO THE FOLLOWING:

```
      if Gas_Pumped > 40.0 then
         Discount_Percent := 10.0;
      else
         if Gas_Pumped > 25.0 then
            Discount_Percent := 5.0;
         else
            Discount_Percent := 0.0;
         end if;
      end if;
```

NOTE THE DECREASE IN READABILITY AND INCREASE IN INDENTATION.

INSTRUCTOR NOTES

THIS IS THE DEFINITION FOUND IN THE LRM.

{ } MEANS elsif CAN OCCUR ZERO OR MORE TIMES

[ ] MEANS THE else PART IS OPTIONAL, BUT CAN APPEAR AT MOST ONCE

SUMMARY

```
if Condition then
   ...
{elsif Condition then
   ... }
[else
   ...]
end if;
```

- SELECTION BY CONDITION
- CONDITION EVALUATES TO False OR True

INSTRUCTOR NOTES

THE case STATEMENT IS ANOTHER CONSTRUCT TO BE USED IN REPRESENTING SELECTION.

CASE STATEMENT

- if STATEMENT SELECTS ON BOOLEAN EXPRESSION

- case STATEMENT SELECTS ON ANY DISCRETE EXPRESSION

- DISCRETE EXPRESSION EVALUATES TO AN
  - INTEGER VALUE
  - ENUMERATION LITERAL

INSTRUCTOR NOTES

POINT OUT THE RESERVED WORDS.

BASIC FORMAT IS PRESENTED FIRST. THE FOLLOWING THREE SLIDES PRESENT VARIATIONS, WITH EXAMPLES AFTER THAT.

NOTE THAT "WHEN OTHERS" IS OPTIONAL. IT IS DISCUSSED ON A LATER SLIDE.

BASIC FORMAT

```
case Discrete_Expression is
   when Choice_1 => statement(s);
   when Choice_2 => statement(s);

         ...

   when Choice_n => statement(s);
  [when others   => statement(s);]

end case;
```

- Discrete_Expression IS ANY LEGAL DISCRETE EXPRESSION
- Choices REPRESENT MUTUALLY EXCLUSIVE AND EXHAUSTIVE VALUES OF THE SAME TYPE AS THE EXPRESSION
- AT LEAST ONE STATEMENT MUST BE PROVIDED FOR EACH ALTERNATIVE IF NO ACTIONS ARE TO BE PERFORMED FOR A GIVEN ALTERNATIVE, THE null STATEMENT MUST BE PROVIDED :

  when Choice_i => null;

INSTRUCTOR NOTES

BAR (|) INDICATES THAT WHEN THE VALUE OF THE CASE EXPRESSION IS <u>EITHER</u> OF THE LISTED VALUES, THE INDICATED STATEMENT(S) WILL BE PERFORMED.

NOTE THAT MORE THAN TWO CHOICES MAY BE LISTED, WITH BARS SEPARATING ALL CHOICES IN THE LIST

```
when Choice_1 | Choice_2 | Choice_3 | ... =>
```

BAR NOTATION

- CHOICES MAY BE SEPARATED BY A BAR AS FOLLOWS:

```
when Choice_1 | Choice_2 => statement(s);
```

- THE BAR (|) IS READ "OR"

INSTRUCTOR NOTES

THIS IS ANOTHER NOTATION THAT MAY BE USED TO REPRESENT THE CHOICES.

DISCRETE RANGE NOTATION

```
when Choice_i .. Choice_j => statement(s);
```

- THE CHOICES MUST BE CONSECUTIVE VALUES FOR THE case EXPRESSION WITHIN THE RANGE Choice_i TO Choice_j.

- THE STATEMENTS ARE EXECUTED IF THE CHOICE VALUE FALLS IN THE GIVEN RANGE.

INSTRUCTOR NOTES

POINT OUT THAT THIS IS A CATCH-ALL.

WHEN OTHERS NOTATION

```
when others => statement(s);
```

- THE CHOICE when others APPEARS LAST AND INCLUDES ALL POSSIBLE VALUES NOT SPECIFIED EXPLICITLY IN THE PREVIOUS CHOICES.

INSTRUCTOR NOTES

POINT OUT THAT SINCE A DISCRETE EXPRESSION IS REQUIRED, YOU COULD NOT WRITE THE EARLIER if STATEMENT WITH THE FLOATING POINT OBJECT Gas_Pumped AS A case STATEMENT.

POINT OUT THAT IF

```
type Day_Type is (Sun, Mon, Tue, Wed, Thur, Fri, Sat);
Day:  Day_Type;
```

THEN

```
case Day is
      when Sat .. Sun => Relax;       --** Illegal
      when Mon .. Fri => Work;
end case;
```

IS ILLEGAL! BECAUSE OF WAY Day_Type IS DECLARED.

SUMMARY OF CASE STATEMENT RULES

- IN A case STATEMENT THE EXPRESSION MUST BE DISCRETE (if STATEMENT SELECTS ON BOOLEAN)
- THERE MUST BE ONE, AND ONLY ONE, POSSIBLE ALTERNATIVE FOR EACH POSSIBLE VALUE OF THE EXPRESSION
- ALL POSSIBLE VALUES MUST BE ACCOUNTED FOR
- CHOICES MAY BE SEPARATED BY A BAR (|)
- CHOICES MAY BE SPECIFIED IN A DISCRETE RANGE
- when others IS OPTIONAL YET WHEN APPEARING MUST BE LAST AND APPEAR ALONE
- IF NO STATEMENTS ARE TO BE EXECUTED FOR A GIVEN ALTERNATIVE, THE null STATEMENT MUST BE SPECIFIED

INSTRUCTOR NOTES

- POINT OUT USE OF RANGE (..) AND BAR (|).

- NOTE THAT "when others" COULD BE USED AS THE LAST OPTION, BUT THE FORM SHOWN IS MORE INFORMATIVE.

  - "when others" IS USEFUL WHEN THERE ARE MANY CHOICES LEFT TO ACCOUNT FOR (SEE NEXT SLIDE)

CASE STATEMENT EXAMPLES

CONTEXT:

```
type Precedence_Type is (Routine, Priority, Immediate, Flash, ECP, Critical);

Message_Precedence : Precedence_Type;

type Channel_Type is (Channel_1, Channel_2, Channel_3);

procedure Transmit_Message (Over : Channel_Type);
```

EXAMPLE:

```
case Message_Precedence is

   when Routine =>

      Transmit_Message (Over => Channel_1);

   when Priority .. Flash =>

      Transmit_Message (Over => Channel_2);

   when ECP | Critical =>

      Notify_Operator ("Receiving high-precedence message!");

      Transmit_Message (Over => Channel_3);

end case;
```

INSTRUCTOR NOTES

"others" MUST BE THE LAST CHOICE.

IF "others" IS NOT PRESENT, ALL POSSIBLE VALUES OF THE TYPE OF THE case EXPRESSION MUST BE COVERED BY THE ALTERNATIVES.

"others" COVERS VALUES THAT ARE NOT EXPLICITLY COVERED.

CASE STATEMENT EXAMPLES - (Continued)

CONTEXT:

```
type Security_Classification is (Unclassified, Restricted, Confidential, Secret,
                                 Top_Secret, Special_Category);
Message_Classification : Security_Classification;
Code_Letter : Character;
```

EXAMPLE:

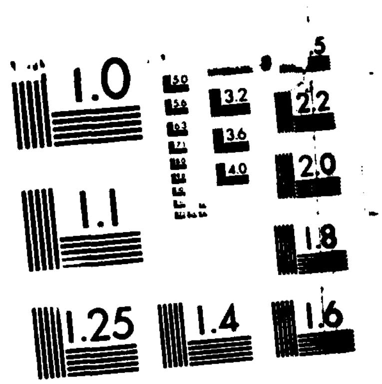
```
case Code_Letter is
   when 'U' =>
      Message_Classification := Unclassified;
   when 'R' =>
      Message_Classification := Restricted;
   when 'C' =>
      Message_Classification := Confidential;
   when 'S' =>
      Message_Classification := Secret;
   when 'T' =>
      Message_Classification := Top_Secret;
   when 'A' =>
      Message_Classification := Special_Category;
   when others =>
      Notify_Operator ("Security Mismatch");
end case;
```

INSTRUCTOR NOTES

EXPECT STUDENTS TO ASK FOR GUIDELINES ABOUT THE USE OF if AND case STATEMENTS. OFTEN IT IS A MATTER OF TASTE. THE BOTTOM LINE IS READABILITY AND MAINTAINABILITY. USING A case STATEMENT FOR DISCRETE TYPES IS OFTEN MORE READABLE BECAUSE OF INDENTATION CONVENTIONS. IT ALSO FORCES THE PROGRAMMER TO CONSIDER WHAT ARE THE APPROPRIATE ACTIONS TO TAKE FOR ALL POSSIBLE VALUES. THE if STATEMENT IS MORE CONCISE WHEN ONLY A SINGLE DISCRETE VALUE IS OF INTEREST. IT MUST BE USED WHEN VALUES BEING CHECKED ARE NOT OF A DISCRETE TYPE, I.E., RECORD, FLOAT, FIXED, ARRAY, ETC.

AD-A166 366 ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA PROGRAMMING L202 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 5/8

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

if STATEMENT VERSUS case STATEMENT

- case STATEMENT
  - DISCRETE TYPES ONLY (EXPRESSION MUST BE OF ENUMERATION OR INTEGER TYPE)
  - MANY ALTERNATIVES LEADING TO DIFFERENT ACTIONS
- if STATEMENT
  - Boolean CONDITION IS THE RESULT OF EVALUATING AN EXPRESSION WHOSE OPERANDS ARE NOT NECESSARILY OF A DISCRETE TYPE
  - "ONE SHOT DEALS" (ONLY ONE VALUE NEEDS TO BE CHECKED)

INSTRUCTOR NOTES

THE SAME PROCEDURE IS USED FOR ENUMERATION I/O AS FOR NUMERIC I/O.

I/O FOR ENUMERATION TYPES

- STEP 1 (AT TOP OF COMPILATION UNIT)

```
with Text_IO; use Text_IO;
```

- STEP 2 (AMONG DECLARATIONS)

```
-- assuming File_Privilege_Type is an enumeration
--      type declared as
-- type File_Privilege_Type is (Read, Write, Edit, Delete);
package File_Priv_IO is new Enumeration_IO
        (File_Privilege_Type);
use File_Priv_IO;
```

- MAKES AVAILABLE:

  Get

  Put

  FOR OBJECTS OF TYPE File_Privilege_Type

INSTRUCTOR NOTES

POINT OUT TWO DIFFERENT I/O PACKAGES USED, ONE FOR EACH ENUMERATION TYPE.

POINT OUT USE OF ATTRIBUTES.

NOTE: Delta_1 IS USED HERE BECAUSE delta IS A RESERVED WORD.

EXAMPLE

```
with Text_IO; use Text_IO;                                              -- STEP 1
procedure Interpret_Classification_Code is
   type Classification_Code_Type is (Alpha, Beta, Gamma,
                                     Delta_1, Omega);
   type Classification_Type is (Top_Secret, Secret, Confidential,
                                Restricted, Unclassified);

   package Class_Code_IO is new Enumeration_IO (Classification_Code_Type);  -- STEP 2
   package Classification_IO is new Enumeration_IO (Classification_Type);   -- STEP 2
   use Class_Code_IO;                                                     -- STEP 2
   use Classification_IO;                                                 -- STEP 2

   Class_Code   : Classification_Code_Type;
   Actual_Class : Classification_Type;
   Position     : Integer;
begin  -- Interpret_Classification_Code
   Get(Class_Code);
   Position := Classification_Code_Type'Pos(Class_Code);
   Actual_Class := Classification_Type'Val(Position);
   Put(Actual_Class);
end Interpret_Classification_Code;
```

SOLUTION:

```
with Text_IO; use Text_IO;
procedure Promote is
   type Army_Rank_Type is ...

   package Rank_IO is new Enumeration_IO (Army_Rank_Type);
   use Rank_IO;
   Current_Rank, New_Rank : Army_Rank_Type;

begin  -- Promote
   Get (Current_Rank);

   if Current_Rank = Captain then
      Put ("No higher rank");
   else
      New_Rank := Army_Rank_Type'Succ(Current_Rank);
      Put (New_Rank);
   end if;
end Promote;
```

- STUDENTS MAY BE TEMPTED TO USE A case STATEMENT TO HANDLE EACH RANK VALUE. POINT OUT HOW MUCH MORE EFFICIENT AND CONCISE THE ATTRIBUTE SOLUTION IS.

- ASSIGN EXERCISES 5, 6, 7 AND 8. IF TIME IS SHORT, EMPHASIZE THAT STUDENTS DO EXERCISES 6 AND 8.

- ASSIGN CHAPTER 5 OF THE PRIMER.

EXERCISE

EXERCISE:

FILL IN THE BLANK PARTS OF THE Promote procedure GIVEN BELOW. THIS PROCEDURE SHOULD READ IN A Rank, DETERMINE THE NEXT-HIGHEST Rank, AND OUTPUT THE NEW Rank. NOTE THAT Captain CAN'T BE PROMOTED FURTHER, SO A MESSAGE SHOULD BE PRINTED IF Captain IS ENTERED.

```
______________________; ______________________;
procedure Promote is
   type Army_Rank_Type is (Pvt, Sergeant, Lieutenant, Captain);

   package Rank_IO is ______________________;
   ______________________;
   Current_Rank, New_Rank : Army_Rank_Type;
begin -- Promote




end Promote;
```

INSTRUCTOR NOTES

ALLOCATE AT LEAST ONE AND ONE HALF HOURS FOR LECTURE ON THIS SECTION.

ASSIGN EXERCISES 9, 10, AND 11 OF THE EXERCISE BOOKLET FOR LAB ASSIGNMENTS AFTER NUMERIC I/O SLIDE.

THE OBJECTIVE OF THIS SECTION IS TO INTRODUCE USER DEFINED NUMERIC TYPES, INTRODUCE OPERATIONS ON OBJECTS OF USER DEFINED NUMERIC TYPES, AND INTRODUCE SUBTYPES OF USER DEFINED NUMERIC TYPES.

SECTION 6
NUMERIC TYPES

INSTRUCTOR NOTES

NUMERIC TYPES

- INTEGER TYPES
  - INCLUDE THE PREDEFINED TYPE Integer
  - HAVE NO DECIMAL POINT
  - HAVE POSITIVE INTEGER EXPONENT ONLY
- REAL TYPES
  - ARE FLOATING POINT TYPES OR FIXED POINT TYPES
    (INCLUDE THE PREDEFINED TYPE Float)
  - MUST HAVE DECIMAL POINT
  - HAVE POSITIVE OR NEGATIVE INTEGER EXPONENTS

INSTRUCTOR NOTES

EXPLAIN THAT IMPLEMENTATION MAY HAVE SEVERAL UNDERLYING TYPES, E.G., Long_Integer, Short_Integer, ETC. COMPILER HAS SEVERAL IMPLEMENTATIONS AVAILABLE - IT WILL SELECT THE MOST APPROPRIATE.

USER-DEFINED INTEGER TYPES

- THE PREDEFINED TYPE Integer BELONGS TO THE <u>CLASS</u> OF INTEGER TYPES

- USER CAN DEFINE Integer TYPES (THE WORD Integer DOES NOT APPEAR IN A USER-DEFINED INTEGER TYPE DELARATION)

```
type Page_Number_Type is range 1 .. 2000;
```

- THE IMPLEMENTATION CHOOSES THE REPRESENTATION OF THE TYPE, BASED ON THE RANGE.

- CONTEXT:

```
Page_Number : Page_Number_Type;
```

- EXAMPLE:

```
Page_Number := Page_Number + 1;
```

INSTRUCTOR NOTES

A STATIC EXPRESSION IS ONE WHOSE VALUE IS KNOWN AT COMPILE TIME.

USER-DEFINED INTEGER TYPES

SYNTAX

```
type Identifier is range Lower_Bound ..Upper_Bound;
```

WHERE

- range Lower_Bound .. Upper_Bound IS CALLED RANGE CONSTRAINT
- RANGE CONSTRAINT IS MANDATORY
- Lower_Bound AND Upper_Bound MUST BE A STATIC EXPRESSION
- Lower_Bound AND Upper_Bound MUST BE OF SOME INTEGER TYPE

INSTRUCTOR NOTES

"FIXED POINT DOES NOT MEAN Integer, AS IN PL/I OR FORTRAN. FIXED POINT OBJECTS HAVE A FIXED WORD LENGTH AND AN ASSUMED DECIMAL POINT (AND THUS A LIMITED RANGE).

IN TYPICAL IMPLEMENTATIONS, FIXED POINT OPERATIONS ARE GENERALLY SIGNIFICANTLY FASTER THAN FLOATING POINT, WHILE PRESERVING THE CORRESPONDENCE WITH THE USER'S PERCEPTION OF THE APPLICATION."

REAL TYPES

ARE APPROXIMATE AND INTRODUCE PROBLEMS OF ACCURACY

- FLOATING POINT
  - HAVE RELATIVE ERROR BOUNDS
- FIXED POINT
  - HAVE ABSOLUTE ERROR BOUNDS

A REAL LITERAL ALWAYS HAS A DECIMAL POINT IN IT

INSTRUCTOR NOTES

THIS FOIL AND THE NEXT TWO FOILS ARE TO SIMPLY HELP THE STUDENT UNDERSTAND THE DIFFERENCE BETWEEN RELATIVE AND ABSOLUTE ERRORS. DO NOT GET BOGGED DOWN IN A DISCUSSION CONCERNING NUMERICAL ANALYSIS.

ERRORS

ABSOLUTE AND RELATIVE

NUMERICAL CALCULATIONS NORMALLY RESULT IN AN APPROXIMATION TO TRUE VALUES BEING SOUGHT. THE ERROR IN THIS APPROXIMATION IS A MEASURE OF DISCREPANCY BETWEEN THE TRUE VALUE AND THE COMPUTED RESULT. BOUNDS OR ESTIMATES OF THE ERROR ARE USUALLY EXPRESSED IN EITHER ABSOLUTE OR RELATIVE FORM.

INSTRUCTOR NOTES

SHOULD A STUDENT POINT OUT THAT IN

AE = TV-APP

THAT ASSIGNMENT IN Ada IS := AND NOT =, INDICATE THAT THIS IS TO INTRODUCE THE STUDENT TO THE DIFFERENCE BETWEEN RELATIVE AND ABSOLUTE ERROR AND IS NOT WRITTEN IN Ada.

ERRORS

ABSOLUTE AND RELATIVE

GIVEN:

TV	true value of quantity
APP	approximate value
AE	absolute error
RE	relative error

THEN MATHEMATICALLY

AE = TV - APP

RE = (TV - APP)/TV = AE/TV

NOTE: NOT Ada SYNTAX

INSTRUCTOR NOTES

IF STUDENTS REQUIRE MORE DETAIL THAN THIS, POINT OUT THAT THIS IS A COURSE IN INTRODUCTORY Ada, NOT A COURSE IN NUMERICAL ANALYSIS!

ERRORS

ABSOLUTE AND RELATIVE

1/3 APPROXIMATED BY .333 THEN

$AE = 1/3 - .333 = 1/3 \times 10^{-3}$ (WHATEVER LEFT OVER)

$RE = (1/3 \times 10^{-3})/(1/3) = 10^{-3}$

SEE: TUTORIAL ON REAL DATA TYPES BY WICHMANN

INSTRUCTOR NOTES

1. "PRECISION MATTERS, ESPECIALLY WHEN ADDING OR SUBTRACTING LARGE NUMBERS TO SMALL NUMBERS."

2. IF CLASS IS SOPHISTICATED, DISCUSS DIFFERENCE BETWEEN ACCURACY AND PRECISION.

   PRECISION : NUMBER OF SIGNIFICANT FIGURES YOU'RE USING (USER'S CHOICE)
   ACCURACY : NUMBER OF SIGNIFICANT FIGURES GUARANTEED TO BE CORRECT. (CONSTRAINT IMPOSED BY MEASUREMENT OR CALCULATION).

   IN GENERAL, PRECISION IS SELECTED SO THAT ONLY LAST SIGNIFICANT DIGIT IS IN DOUBT, E.G.:

   $\underbrace{12.34}_{\text{certain}}5$ accuracy of measurement or calculation makes last digit less certain

   $\underbrace{12.345}_{\text{certain}}\underbrace{6789}_{\text{wasted}}$

   uncertain

   POINT OUT THAT IF TYPE My_Real IS DIGITS 2, 100,000 CAN BE REPRESENTED AS 1.0E6 WHEREAS 1,000,001 CANNOT BE REPRESENTED.

   POINT OUT THAT IF MORE DIGITS ARE SPECIFIED FOR AN OBJECT THAN IS DEFINED IN THE TYPE DECLARATION, THE IMPLEMENTATION MAY OR MAY NOT HANDLE IT. IF IT CAN, ITS OK, BUT THE LANGUAGE DOESN'T REQUIRE IT.

REAL -- FLOATING POINT

SYNTAX:

```
type Identifier is digits N [range Lower_Bound .. Upper_Bound];
```

NOTE: N REPRESENTS THE MINIMUM NUMBER OF SIGNIFICANT DECIMAL DIGITS.

N MUST BE AN INTEGER.

EXAMPLES:

```
type Scores is digits 5 range 0.0 .. 1500.0;
type Sensor_Reading_Type is digits 5;
type Sales_Type is digits 9 range 0.0 .. 1_000_000.0;
type Probability_Type is digits 4 range 0.0 ... 1.0;
```

INSTRUCTOR NOTES

POINT OUT THAT WHEREAS RANGE CONSTRAINT IS OPTIONAL IN FLOATING POINT, IT IS REQUIRED IN FIXED POINT.

POINT OUT THAT IF AN OBJECT Battery_Voltage OF TYPE Voltage_Type HAD BEEN ASSIGNED 11.8, IT IS IMPLEMENTATION DEPENDENT WHAT VALUE WOULD BE STORED INTERNALLY. IT COULD ROUND UP OR DOWN.

(IN FACT, EXACT REPRESENTATIONS SHOULD NOT BE DEPENDED ON, EVEN FOR EXACT MULTIPLES OF THE DELTA. GIVEN

```
F: Fraction_Type := 0.01;
```

THE INTERNAL VALUE ASSIGNED TO F MAY BE AS LOW AS 1/128 (0.0078125) OR AS HIGH AS 2/128 (0.015625). IT IS GUARANTEED TO BE LESS THAN 0.01 AWAY FROM THE SPECIFIED INITIAL VALUE.)

REAL -- FIXED POINT

SYNTAX:

```
type Identifier is delta D range Lower_Bound .. Upper_Bound;
```

NOTE: D REPRESENTS THE DEGREE OF ACCURACY AND MUST BE A POSITIVE REAL NUMBER

THE RANGE CONSTRAINT IS REQUIRED

EXAMPLES:

```
type Voltage_Type is delta 0.125 range -12.0 .. 12.0;
type Temperature_Type is delta 0.5 range 70.0 .. 110.0;
type Velocity_Type is delta 1.0 range 0.0 .. 85.0;
type Weight_Type is delta 0.5 range 0.0 .. 200.0;
Interval: constant := 0.01;
type Fraction_Type is delta Interval range 0.0 .. 1.0 - Interval;
Battery_Voltage : Voltage_Type := 11.875;
```

INSTRUCTOR NOTES

REAL TYPES ALSO HAVE ATTRIBUTES.

REAL ATTRIBUTES

FX'Delta	THE VALUE SPECIFIED IN THE ACCURACY DEFINITION OF A FIXED POINT TYPE
FL'Digits	THE NUMBER OF SIGNIFICANT DIGITS IN A FLOATING POINT TYPE
R'First	LOWER BOUND OF R
R'Last	UPPER BOUND OF R

WHERE FX IS A FIXED-POINT TYPE, FL IS A FLOATING-POINT TYPE, AND R IS EITHER.

CONTEXT:

```
type Real is digits 8;
type Percentage_Type is digits 4;
type Temperature_Type is delta 0.5 range 70.0 .. 110.0;
Digit_Count : Integer;
Temperature_Delta : Float;
```

EXAMPLES:

```
Digit_Count := Real'Digits;        -- 8
if Percentage_Type'Digits < 5 then -- True
   Temperature_Delta := Temperature_Type'Delta;     -- 0.5
end if;
```

INSTRUCTOR NOTES

OBJECT DECLARATIONS ARE AS FOR OTHER TYPES.

OBJECT DECLARATIONS

CONTEXT:

```
type Page_Number_Type is range 1 .. 2000;
type Scores is digits 5 range 0.0 .. 1500.0;
type Temperature_Type is delta 0.5 range 70.0 .. 110.0;
```

EXAMPLE:

```
Page_Number :  Page_Number_Type := 1;
Score       :  Scores := 99.5;
Temperature :  Temperature_Type;
```

INSTRUCTOR NOTES

A REMINDER.

REMEMBER

- IN Ada, ITEMS OF DATA ARE CALLED OBJECTS.
- IN ADDITION TO A VALUE, EVERY OBJECT HAS A PROPERTY CALLED TYPE, WHICH CONSTRAINS THE FORMS THAT THE VALUE MAY TAKE AND THE OPERATIONS THAT MAY BE APPLIED TO THE OBJECT.
- THE TYPE OF EACH OBJECT MUST BE SPECIFIED IN THE DECLARATIVE PART OF THE PROGRAM.

INSTRUCTOR NOTES

"BLANK SPACES IN THE TABLE MEAN THAT THE LANGUAGE DOES NOT DEFINE THE OPERATION FOR OPERANDS OF THE SPECIFIED TYPE(S). THERE IS A WAY, DISCUSSED LATER, TO DEFINE ADDITIONAL MEANINGS FOR OPERATORS. THIS TABLE EXPLAINS THE PREDEFINED MEANINGS."

I CAN BE READ AS "ANY INTEGER TYPE," FL AS "ANY FLOATING POINT TYPE," FX1 AS "ANY FIXED POINT TYPE," AND FX2 AS "ANY OTHER FIXED POINT TYPE."

THE FX1 op FX2 COLUMN REPRESENTS OPERATIONS THAT CAN BE APPLIED TO OPERANDS IN DIFFERENT FIXED POINT TYPES.

THE FX1 op FX1 COLUMN REPRESENTS OPERATIONS THAT CAN BE APPLIED ONLY TO OPERANDS IN THE SAME FIXED POINT TYPE.

THE LAST THREE COLUMNS INVOLVE OPERATIONS IN WHICH AN OPERAND BELONGS NOT TO ANY INTEGER TYPE IN GENERAL, BUT TO THE SPECIFIC PREDEFINED TYPE Integer.

EXAMPLE OF FIXED-POINT MULTIPLICATION:

```
type Fix is delta 0.1 range 1.0 .. 50.0;
   A, B, C : Fix := 5.0;
begin
   C := A*B;          -- ILLEGAL TYPE MISMATCH.
   C := Fix (A*B);  -- LEGAL
end;
```

INDIVIDUAL OPERATORS, INCLUDING rem AND mod, ARE EXPLAINED IN GREATER DETAIL ON LATER SLIDES.

RESULTS OF NUMERIC OPERATIONS

LEGEND:

I	-- an integer type
FL	-- a floating point type
FX1	-- one fixed point type
FX2	-- another fixed point type
B	-- Boolean

	op	op I	I op I	op FL	FL op FL	op FX1	FX1 op FX2	FX1 op FX1	FL op Integer	FX1 op Integer	Integer op FX1
highest precedence	**		I#						FL		
	abs	I		FL		FX1					
multiplying	*		I		FL		*	*		FX1	FX1
	/		I		FL		*	*		FX1	
	mod		I								
	rem		I								
unary adding	unary +	I		FL		FX1					
	unary -	I		FL		FX1					
binary adding	binary +		I		FL			FX1			
	binary -		I		FL			FX1			
relational	=		B		B			B			
	/=		B		B			B			
	<		B		B			B			
	<=		B		B			B			
	>		B		B			B			
	>=		B		B			B			

*MULTIPLICATION OR DIVISION OF TWO FIXED-POINT VALUES CAN ONLY TAKE PLACE INSIDE A TYPE CONVERSION. THE TYPE CONVERSION DETERMINES THE TYPE OF THE RESULT.
#EXPONENT MUST BE NON NEGATIVE FOR INTEGER TYPES.

INSTRUCTOR NOTES

"IF THE TABLE ON 6-13 DOES NOT PROVIDE FOR THE OPERAND TYPES YOU ARE USING, TYPE CONVERSIONS MAY BE APPLIED TO ONE OR BOTH OPERANDS."

A CONVERSION FROM A REAL TYPE TO AN INTEGER TYPE MAY REQUIRE ROUNDING TO THE NEAREST WHOLE NUMBER. A CONVERSION FROM ONE REAL TYPE TO ANOTHER MAY RESULT IN A CHANGE OF INTERNAL REPRESENTATION AND LOSS OF PRECISION, BUT THE VALUE REPRESENTED REMAINS APPROXIMATELY THE SAME.

NUMERIC TYPE CONVERSIONS

SYNTAX:

```
type_name (expression)
```

CONTEXT:

```
type Fix is delta 0.01 range 0.0 .. 1.0;
type Volume_In_Liters is digits 6;
I  : Integer;
FX : Fix;
FL : Float;
V  : Volume_In_Liters;
```

EXAMPLES:

```
I  := I + Integer (FX);
FL := Float (FX) * Float (I);
V  := Volume_In_Liters (FL) * V;  -- Conversion needed to multiply
                                  -- values in different floating
                                  -- point types.
```

INSTRUCTOR NOTES

POINT OUT THAT USER DEFINED NUMERIC TYPES ARE SUBJECT TO THE SAME OPERATORS AND HIERARCHY RULES AS PREDEFINED NUMERIC TYPES.

"WHEN OPERATORS ARE COMBINED IN AN EXPRESSION, THERE ARE SPECIFIC RULES REGARDING WHICH OPERATORS ARE APPLIED FIRST. IN GENERAL, THOSE OF HIGHEST PRECEDENCE ARE APPLIED, FOLLOWED BY THOSE OF SUCCESSIVELY LOWER PRECEDENCE. THESE RULES GREATLY REDUCE THE NEED FOR PARENTHESES, MAKING EXPRESSIONS EASIER TO READ, SINCE THE PRECEDENCE RULES CLOSELY FOLLOW THOSE OF MATHEMATICS.

OF COURSE, WHERE PARENTHESES ARE USED, THEY OVERRIDE THE RULES OF PRECEDENCE. OFTEN, JUDICIOUS USE OF PARENTHESES, EVEN WHEN NOT ACTUALLY NECESSARY CAN MAKE A STATEMENT MORE READABLE."

"WHEN OPERATORS OF THE SAME PRECEDENCE OCCUR IN AN EXPRESSION, THEY ARE APPLIED FROM LEFT TO RIGHT, EXCEPT THE LOGICAL OPERATORS, WHICH REQUIRE PARENTHESES." (THIS EXCEPTION IS CAREFULLY CONCEALED IN SECTION 4.4 (AND SEEMINGLY CONTRADICTED IN 4.5) OF THE Ada LANGUAGE REFERENCE MANUAL).

EXPLAIN EXCLUSIVE OR (xor).

"DON'T CONFUSE & WITH and."

HIERARCHY OF OPERATORS

Highest Precedence

↓

Lowest Precedence

parentheses	( )				
highest precedence	**	abs	not		
multiplying	*	/	mod	rem	
unary	+	-			
binary	+	-	&		
relational/membership	=	/=	<	<=	>
	>=	in	not in		
logical	and	or	xor	and then	
	or else				

INSTRUCTOR NOTES

ADDITIONAL RESTRICTIONS

- IF AN OPERAND OF abs OR ** IS AN EXPRESSION THAT CONTAINS ANY OPERATOR, THAT OPERAND MUST BE ENCLOSED IN PARENTHESES.

  CONTEXT:

  ```
  A, B, C:  Integer;
  ```

  EXAMPLES:

  ```
  A ** B ** C IS ILLEGAL.   WRITE (A ** B) ** C OR A ** (B ** C).
  abs A ** B IS ILLEGAL.    WRITE (abs A) ** B OR abs(A ** B).
  A ** B+C   IS ILLEGAL.    WRITE A ** (B+C).
  ```

- INTEGER VERSION OF EXPONENTIATION

  EXPONENT MUST BE NON-NEGATIVE

INSTRUCTOR NOTES

"abs IS A UNARY OPERATOR LIKE UNARY MINUS."

THE abs OPERATOR

ABSOLUTE VALUE EXAMPLES:

abs 3 = 3

abs (-25) = 25

abs 0 = 0

INSTRUCTOR NOTES

INTEGER DIVISION

WHEN / IS APPLIED TO TWO OPERANDS IN AN INTEGER TYPE, THE RESULT IS TRUNCATED TOWARDS ZERO IF NECESSARY TO PRODUCE A RESULT IN THE SAME TYPE.

4/2 = 2

7/4 = 1

-4/2 = -2

-7/4 = -1

REMEMBER: TYPE CONVERSION TO AN INTEGER TYPE ROUNDS TO THE NEAREST INTEGER.

INTEGER DIVISION TRUNCATES TOWARDS ZERO.

Integer (7.0/4.0) = Integer (1.75) = 2

Integer (7.0)/Integer (4.0) = 7/4 = 1

INSTRUCTOR NOTES

Mod IS DISCUSSED IN THE NEXT FOIL. TWO VIEWGRAPHS AWAY IS A GRAPH COMPARING mod AND rem.

THE rem OPERATOR

PRODUCES REMAINDER CORRESPONDING TO INTEGER DIVISION. TRUNCATES TOWARD ZERO.

EXAMPLE:

```
17 rem 3 = 2

        dividend is 17
        divisor is 3
        quotient is 5

remainder = dividend - (quotient * divisor)
                 = 17 - (5 * 3)
                 = 17 - 15
                 = 2
```

EXAMPLE:

```
-7 rem 3 = -1
        dividend is -7
        divisor is 3
        quotient is -2
remainder = dividend - (quotient * divisor)
        = -7 - (-2 * 3)

        = -7 + 6

        = -1
```

SIGN OF RESULT IS ALWAYS EQUAL TO SIGN OF THE FIRST ARGUMENT WHICH IS THE DIVIDEND.

INSTRUCTOR NOTES

RELATE mod TO EQUIVALENCE CLASSES.

THE mod OPERATOR

REMAINDER CORRESPONDING TO DIVISION WITH ROUNDING TOWARD MINUS INFINITY.

NOTE: rem AND mod DIFFER ONLY WHEN ONE OPERAND IS NEGATIVE.

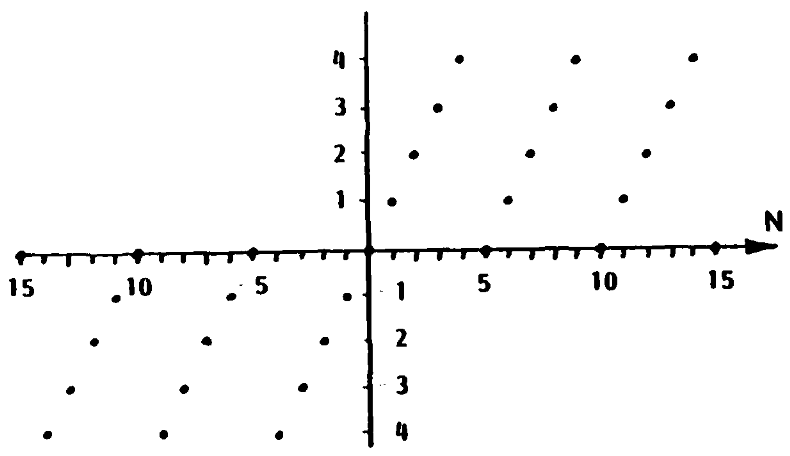
EXAMPLE:

```
-7 mod 3 = 2

        dividend is -7
        divisor is 3
        quotient is -3 (rounds toward minus infinity)

mod is

        -7 - (-3 * 3)

        -7 + 9

        2
```

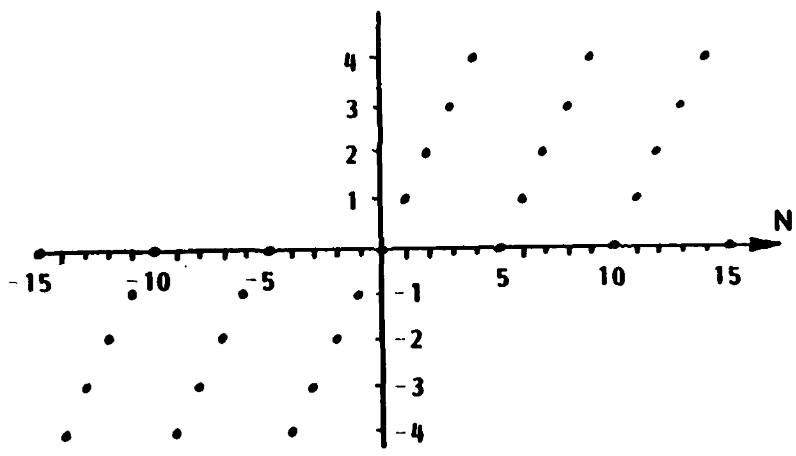
INSTRUCTOR NOTES

POINT OUT THAT ONLY THE FIRST QUADRANT IS GENERALLY USED (ARGUMENT AND MODULUS BOTH POSITIVE). AVOID LONG DISCUSSION (THAT'S WHY THE GRAPHS ARE INCLUDED).

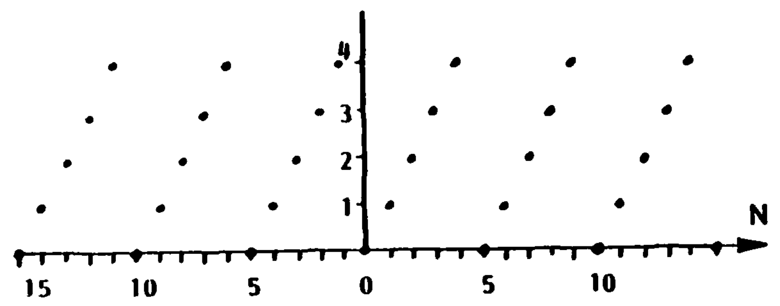
rem AND mod GRAPHS



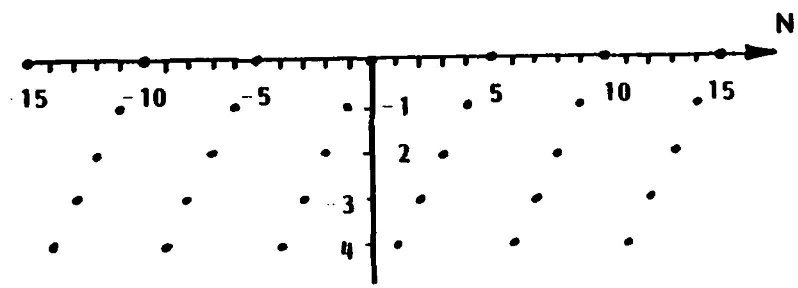
N rem 5



N rem -5



N mod 5



N mod -5

INSTRUCTOR NOTES

RELATIONAL OPERATORS

=   /=   <   <=   >   >=

RETURN RESULTS OF TYPE Boolean

INSTRUCTOR NOTES

THIS STEP BY STEP APPROACH WILL BE USED FOR EACH TYPE.

EXPLAIN THAT STEP 2 IS FULLY EXPLAINED IN L305. GIVE THE STUDENTS A BRIEF EXPLANATION HERE OF GENERICS. DON'T GET BOGGED DOWN.

NUMERIC I/O

STEP 1: AT TOP OF PROGRAM:

```
with Text_IO; use Text_IO;
```

STEP 2: INSIDE PROGRAM (AMONG THE DECLARATIONS):

- for an integer type My_Integer:

  ```
  type My_Integer is range 10 .. 50;
  ```

  Write:

  ```
  package My_Integer_IO is new Integer_IO (My_Integer);
  use My_Integer_IO;
  ```

- for a floating-point type My_Float:

  ```
  type My_Float is digits 3;
  ```

  Write:

  ```
  package My_Float_IO is new Float_IO (My_Float);
  use My_Float_IO;
  ```

- for a fixed-point type My_Fixed:

  ```
  type My_Fixed is delta 0.1 range 1.0 .. 20.0;
  ```

  Write:

  ```
  package My_Fixed_IO is new Fixed_IO (My_Fixed);
  use My_Fixed_IO;
  ```

YOU GET

- Put
- Get

INSTRUCTOR NOTES

POINT OUT THAT STEP 2 MUST BE REPEATED FOR EACH USER DEFINED TYPE.

AN EXAMPLE

```
with Text_IO; use Text_IO;                                        -- Step 1
procedure Calculate_Average_Yield is

  type Bushels_Type is range 0 .. 100;                             -- user defined Integer-type
  type Average_Bushels_Type is digits 5 range 0.0 .. 100.0;        -- user defined Floating Type
  Bushels_1, Bushels_2, Bushels_3: Bushels_Type;                   -- objects
  Average_Bushels: Average_Bushels_Type;                           -- object
  package Bushels_IO is new Integer_IO (Bushels_Type);             -- I/O for Bushels_Type Step 2
  package Average_IO is new Float_IO (Average_Bushels_Type);       -- I/O for Average_Bushels_Type
  use Bushels_IO; use Average_IO;                                  -- Step 2

begin -- Calculate_Average_Yield

  Put_Line ("Enter Bushels for Field 1 : ");
  Get (Bushels_1);
     ...
  Average_Bushels := Average_Bushels_Type (Bushels_1 + Bushels_2 + Bushels_3)/3.0;
  Put (Average_Bushels);

end Calculate_Average_Yield;
```

INSTRUCTOR NOTES

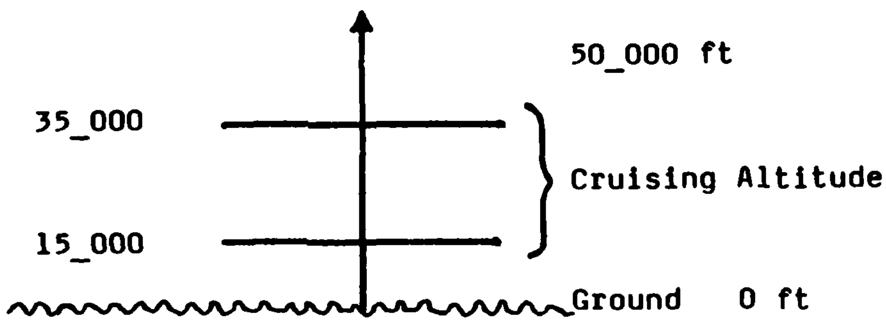
DIAGRAM SHOWS THE ALTITUDE, 0 TO 50_000 FEET. THE CRUISING-ALTITUDE IS A FURTHER CONSTRAINT OF ALTITUDE.

SUBTYPES

- RANGE CONSTRAINTS PUT LOWER AND UPPER BOUNDS ON ACCEPTABLE VALUES FOR OBJECTS OF THAT TYPE

```
type Altitude_Type is range 0 .. 50_000;
Cruising_Altitude : Altitude_Type range 15_000 .. 35_000;
```

- TO PUT ADDITIONAL CONSTRAINTS ON *SUBSET OF OBJECTS USE SUBTYPES*



INSTRUCTOR NOTES

POINT OUT THAT IN EVALUATING NUMERIC EXPRESSIONS, INTERMEDIATE RESULTS MAY BE OUT OF RANGE. THE FINAL RESULT IS THE VALUE THE CONSTRAINT IS CHECKED AGAINST.

SUBTYPES

VALUES

- SUBSET OF VALUES OF SOME BASE TYPE
- DEFINED BY MEANS OF A CONSTRAINT

OPERATIONS

- SUBTYPE HAS ALL THE SAME OPERATIONS OF THE BASE TYPE
- INTERMEDIATE RESULTS MAY BE OUTSIDE RANGE

INSTRUCTOR NOTES

POINT OUT THAT Upper_Bound AND Lower_BOUND COULD BE OBJECT NAMES. POINT OUT THAT *Identifiers*, *Type_Name*, Upper_Bound AND Lower_Bound ARE USER SUPPLIED NAMES.

THE RANGE CLAUSE IS OPTIONAL.

GOOD REVIEW QUESTION TO ASK CLASS: HOW DO YOU WRITE A MAINTAINABLE PROGRAM THAT USES THE UPPER AND LOWER BOUNDS OF *Altitude_Type and Cruising_Altitude_Type*?

ANSWER: REFER TO VALUES 500000, 35000 AND 15000 WITH ATTRIBUTES FIRST AND LAST.

SUBTYPE DECLARATION

SYNTAX:

```
subtype Identifier is Type_Name [range Lower_Bound .. Upper_Bound];
```

"Type_Name" IS THE BASE TYPE

"range Lower_Bound .. Upper_Bound" IS CALLED THE RANGE CONSTRAINT.

MAY USE STATIC EXPRESSIONS IN EXPRESSING THE LOWER AND UPPER BOUNDS OF THE RANGE CONSTRAINT.

EXAMPLE:

```
type Altitude_Type is range 0 .. 50_000;
subtype Cruising_Altitude_Type is Altitude_Type range 15_000 .. 35_000;
```

INSTRUCTOR NOTES

THE PURPOSE OF THIS FOIL IS TO INTRODUCE THE TERMINOLOGY BASE TYPE.

SUBTYPES AND BASE TYPES

If *s* IS A SUBTYPE OF TYPE *t*, *t* IS CALLED THE BASE TYPE OF *s*.

CONTEXT:

```
type Altitude_Type is range 0 .. 50_000;
subtype Cruising_Altitude_Type is Altitude_Type range 15_000 .. 35_000;
```

EXAMPLE:

Altitude_Type IS THE BASE TYPE of Cruising_Altitude_Type

VALUES OF THE SUBTYPE:

SUBSET OF THE VALUES OF THE BASE TYPE

(15_000 TO 35_000)

OPERATIONS OF THE SUBTYPE:

SAME AS THE BASE TYPE

(THOSE OPERATIONS AVAILABLE TO CLASS OF INTEGER TYPES)

INSTRUCTOR NOTES

OPERATIONS ON VARIABLES IN A GIVEN SUBTYPE MAY PRODUCE RESULTS THAT ARE OUTSIDE OF THE SUBTYPE. THIS IS OKAY, AS LONG AS VALUES STORED IN THE VARIABLE BELONG TO THE APPROPRIATE SUBTYPE.

SUBTYPE EXAMPLES

CONTEXT:

```
type Scores is digits 5 range 0.0 .. 1500.0;
subtype Scores_Type is Scores range 0.0 .. 100.0;

type List_Type is array (1 .. 15) of Scores_Type;
```

EXAMPLE:

```
function Mean (List : List_Type)
                    return Scores_Type is
   Sum : Scores := 0.0;

begin -- Mean

   for I in List'Range loop

      Sum := Sum + List(I);       -- Combination
                                  -- of type and
   end loop;                      -- subtype.

   return Sum/Scores_Type (List'Last);

end Mean;
```

INSTRUCTOR NOTES

ASK THE CLASS HOW THEY COULD MAKE THIS EXAMPLE LEGAL WITHOUT SUBTYPE. (ANSWER - THEY WOULD NEED TYPE CONVERSION.)

THIS WOULD MAKE THE PROGRAM MORE DIFFICULT TO READ.

SUBTYPE MOTIVATION

CONTEXT:

```
type Line_Length_Type is range 0 .. 1000;
type Word_Length_Type is range 0 .. 30;
Line_Length : Line_Length_Type;
Word_Length : Word_Length_Type;
```

EXAMPLE:

```
Line_Length :=  Line_Length + Word_Length;  -- **ILLEGAL  Line_Length
                                            -- and Word_Length are
                                            -- of different types
```

where Word_Length_Subtype is subtype of Line_Length_Type:

```
Line_Length := Word_Length;  -- always legal
Word_Length := Line_Length;  -- may cause an error, depending on
                             -- the value of Line_Length
```

SUBTYPE MOTIVATION (Continued)

RESTRICT RANGE WITHOUT INTRODUCING NEW TYPES

CONTEXT:

```
type Line_Length_Type is range 0 .. 1000;
Line_Length  :  Line_Length_Type;
subtype Word_Length_Subtype is Line_Length_Type range 0 .. 30;
Word_Length  :  Word_Length_Subtype;
```

EXAMPLE:

```
Line_Length  := Line_Length + Word_Length;     -- OK
```

INSTRUCTOR NOTES

THE POINT IS THAT OBJECTS OF TYPES DECLARED WITH A SPECIFIC CONSTRAINT MAY THEMSELVES BE DECLARED WITH AN ADDITIONAL CONSTRAINT.

RANGE CONSTRAINTS ON VARIABLES

FOR SCALAR TYPES, A RANGE CONSTRAINT CAN FOLLOW THE TYPE OR SUBTYPE NAME IN A VARIABLE DECLARATION.

CONTEXT:

```
type Line_Length_Type is range 0 .. 1000;
```

THE DECLARATION:

```
Word_Length : Line_Length_Type range 0 .. 30 := 0;
```

IS EQUIVALENT TO:

```
subtype Word_Length_Subtype is Line_Length_Type range 0 .. 30;
Word_Length : Word_Length_Subtype := 0;
```

INSTRUCTOR NOTES

"Integer'Last MEANS THE LAST VALUE IN THE SERIES OF ALL POSSIBLE VALUES OF THE TYPE Integer. THIS IS IMPLEMENTATION-DEPENDENT. THE ATTRIBUTE 'Last APPLIES TO ANY SCALAR TYPE."

ASSIGN EXERCISE 9, 10 AND 11 AFTER FINISHING THIS SECTION.

ASSIGN CHAPTER 6 OF THE PRIMER.

PREDEFINED SUBTYPES

```
subtype Positive is Integer range 1 .. Integer'Last;
subtype Natural is Integer range 0 .. Integer'Last;

-- Integer'Last is the highest value of type Integer.
```

EXAMPLES:

```
Age   : Natural range 0 .. 105;
Count : Natural;
```

INSTRUCTOR NOTES

THE OBJECTIVE OF THIS SECTION IS TO INTRODUCE SOME ADDITIONAL FEATURES OF SCALAR TYPES, I.E., SHORT CIRCUIT CONTROL FORMS, RELATIONAL OPERATORS, MEMBERSHIP TESTS, THE TYPE Character, AND ATTRIBUTES.

ASSIGN EXERCISE 12 AT THE END OF THIS SECTION.

SECTION 7
ADDITIONAL FEATURES OF SCALAR TYPES

INSTRUCTOR NOTES

THE Ada LANGUAGE REQUIRES THAT BOTH OPERANDS TO THE and OPERATOR BE EVALUATED, BUT DOES NOT SPECIFY IN WHICH ORDER.

SHORT-CIRCUIT CONTROL FORMS

MOTIVATION

THE LOGICAL OPERATORS: and or xor

COMPARE TWO BOOLEAN EXPRESSIONS, EACH OF WHICH IS EVALUATED SEPARATELY BEFORE THE LOGICAL OPERATOR IS APPLIED. THIS CAN LEAD TO PROBLEMS:

CONTEXT:

```
Time, Distance: Float;
```

EXAMPLE:

```
if (Time /= 0.0) and (Distance/Time < 5.2) then
                    .
                    .
                    .
end if;
```

THE RIGHT HAND EXPRESSION WILL RESULT IN A RUNTIME ERROR IF Time = 0.0. WE WANT NOT ONLY TO TEST FOR THAT CONDITION, BUT TO AVOID EVALUATING THE RIGHT HAND EXPRESSION IF THE LEFT ONE YIELDS False.

INSTRUCTOR NOTES

POINT OUT THAT THE TYPICAL SOLUTION USES NESTED IFS AND Ada OFFERS A BETTER SOLUTION.

A TYPICAL SOLUTION

THE OBVIOUS SOLUTION IS:

CONTEXT:

```
Time, Distance: Float;
```

EXAMPLE:

```
if Time /= 0.0 then
   if Distance/Time < 5.2 then
                     .
                     .
                     .
   end if;
end if;
```

INSTRUCTOR NOTES

SHORT CIRCUIT CONTROL FORMS

THE LANGUAGE PROVIDES TWO "SHORT-CIRCUIT CONTROL FORMS" WHICH AVOID UNNECESSARY AND POSSIBLY INVALID EVALUATION OF EXPRESSIONS:

and then   or else

WITH THESE FORMS, THE LEFT EXPRESSION IS EVALUATED FIRST, AND IF THE RESULT IS SUFFICIENT TO DETERMINE THE OUTCOME, THE RIGHT IS IGNORED. ASSUMING THE SAME CONTEXT,

```
if (Time /= 0.0) and then (Distance/Time < 5.2) then
   ...
end if:
```

HERE, IF THE FIRST EXPRESSION IS False (i.e., Time = 0.0), THIS DETERMINES THE RESULT OF THE ENTIRE EXPRESSION (False).

SIMILARLY, WITH or else A True FIRST EXPRESSION MAKES THE WHOLE EXPRESSION True.

INSTRUCTOR NOTES

POINT OUT THAT IT IS MORE READABLE THAN

```
if Bushels_1 > 75 then
   ...
elsif Bushels_1 > 50 then
   ...
else
   ...
end if;
```

MEMBERSHIP TEST

- in TESTS MEMBERSHIP OF THE RESULT OF AN EXPRESSION IN A TYPE (OR SUBTYPE) OR RANGE.

CONTEXT:

```
type Bushels_Type is range 0 .. 100;
Bushels_1  :  Bushels_Type;
```

EXAMPLE:

```
if Bushels_1 in 0 .. 50 then
      Charge_Rate_1;
elsif Bushels_1 in 51 .. 75 then
      Charge_Rate_2;
else
      Charge_Rate_3;
end if;
```

NOTE:

```
Bushels_1 in 0 .. 50  -- equivalent to
                      -- Bushels_1 <= 50 and
                      -- Bushels_1 >= 0
```

INSTRUCTOR NOTES

MEMBERSHIP TEST

- not in IS THE COMPLEMENT OF in

INSTRUCTOR NOTES

"NOTE THAT AN OBJECT OF TYPE Character HAS A VALUE THAT IS EQUAL TO <u>ONE AND ONLY ONE</u> ASCII CHARACTER. THERE EXISTS A TYPE CALLED String, WHICH IS A STRING OF CHARACTERS, AND WILL BE DISCUSSED LATER UNDER ARRAY TYPES."

IF SOMEONE ASKS, A SINGLE QUOTE IN A STRING IS REPRESENTED "'". DEFER ALL QUESTIONS ON STRING UNTIL SECTION 8.

THE PREDEFINED TYPE Character

- PREDEFINED ENUMERATION TYPE
  - 128 CHARACTERS OF THE ASCII SET
  - 95 PRINTABLE CHARACTERS DENOTED BY CORRESPONDING CHARACTER LITERAL

- SINGLE CHARACTER LITERALS ARE ENCLOSED BY SINGLE QUOTES

'A'	THE CHARACTER A
'5'	THE CHARACTER 5
' '	THE CHARACTER BLANK
'''	THE CHARACTER APOSTROPHE (')

- EXAMPLES:

```
Command_Character  :  Character;
Prompt             :  constant Character := '?';
```

INSTRUCTOR NOTES

DON'T GO INTO DETAILS. THIS FEATURE IS NOT OF GENERAL USE. ITS PRIMARY PURPOSE IS TO ALLOW TYPES CORRESPONDING TO CHARACTER SETS OTHER THAN ASCII (E.G., EBCDIC).

NOTE TO INSTRUCTOR: THE DECLATION OF Roman_Digit OVERLOADS THE CHARACTER LITERAL 'I' (AMONG OTHERS). IN THE DECLARATION

```
One : Roman_Digit_Type := 'I';
```

THE "character literal" 'I' IS NOT OF TYPE CHARACTER. IN FACT, IT HAS NOTHING WHATSOEVER TO DO WITH THE CHARACTER VALUE NAMED BY THE SAME LITERAL. Ada PROVIDES NO CONVENIENT WAY TO ESTABLISH ANY CORRESPONDENCE BETWEEN THE TWO. AN INCONVENIENT WAY IS Character'Value (Roman_Digit'Image(X)), WHICH CONVERTS THE FIRST VALUE OF TYPE Roman_Digit INTO THE STRING VALUE "'I'", AND THEN BACK TO THE (Character'Pos('I'))th VALUE OF TYPE CHARACTER.

ADDITIONAL CHARACTER TYPES

- ENUMERATION LITERALS CAN BE EITHER IDENTIFIERS OR CHARACTER LITERALS.

- ANY ENUMERATION TYPE FOR WHICH AT LEAST ONE POSSIBLE ENUMERATION LITERAL IS A CHARACTER LITERAL IS SAID TO BE A CHARACTER TYPE.

- IT IS POSSIBLE TO DEFINE OTHER CHARACTER TYPES IN ADDITION TO THE PREDEFINED TYPE Character.

EXAMPLES:

```
type Roman_Digit_Type is ('I', 'V', 'X', 'L', 'C', 'D', 'M');

type Security_Code_Type is ('U', 'C', 'S', 'T', 'E');
```

INSTRUCTOR NOTES

OPERATIONS FOR OBJECTS OF Character TYPES ARE THE SAME AS OPERATIONS FOR <u>ALMOST</u> ANY ENUMERATION TYPE. (Boolean HAS LOGICAL OPERATIONS DEFINED FOR IT.)

CHARACTER TYPE OPERATIONS
(SAME AS FOR ANY ENUMERATION TYPE)

- RELATIONAL OPERATIONS
- ASSIGNMENT

INSTRUCTOR NOTES

I/O FOR PREDEFINED TYPE Character

- ONLY STEP 1 REQUIRED. PLACE AT TOP OF COMPILATION UNIT

```
with Text_IO; use Text_IO;
```

- YOU GET:

```
Get (C);      -- where C : Character;
Put (C);
```

  - CONTEXT:

```
Command_Character : Character;
Prompt : constant Character := '?';
```

  - EXAMPLES:

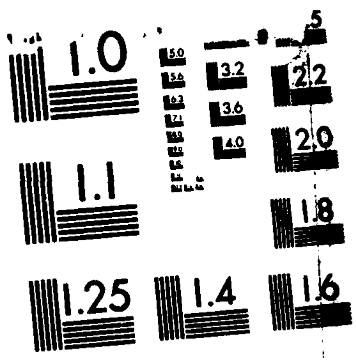
```
Put ('A');

Get (Command_Character);

Put (Prompt);
```

INSTRUCTOR NOTES

THE Ada LANGUAGE USES THE ASCII CHARACTER SET.

AD-A166 366 ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA PROGRAMMING L202 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514 6/8

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NON-PRINTABLE CHARACTERS

- CANNOT BE INCLUDED IN THE TEXT
- PREDEFINED CONSTANTS ARE AVAILABLE IN package ASCII
- NO with CLAUSE IS NEEDED TO USE package ASCII.

EXAMPLE:

```
Put (ASCII.BEL);     -- RINGS THE BELL
```

INSTRUCTOR NOTES

STATE IT IS IMPLEMENTED AS AN ENUMERATION TYPE.

POINT OUT POSITIONING OF UPPERCASE AND LOWER CASE LETTERS.

ASSIGN EXERCISE 12.

ASSIGN CHAPTER 7 OF THE PRIMER.

NOTE THAT ITALICIZED CHARACTERS ARE IMPLEMENTATION-DEPENDENT, WHICH MEANS THAT PRINTING SUCH A CHARACTER INVOLVES A STATEMENT LIKE PUT (ASCII.BEL) RATHER THAN PUT (BEL). THIS INSURES A MEASURE OF PORTABILITY.

type CHARACTER is

```
(nul,   soh,   stx,   etx,      eot,   enq,   ack,   bel,
 bs,    ht,    lf,    vt,       ff,    cr,    so,    si,
 dle,   dcl,   dc2,   dc3,      dc4,   nak,   syn,   etb,
 can,   em,    sub,   esc,      fs,    gs,    rs,    us,

 ' ',   '!',   '"',   '#',      '$',   '%',   '&',   ''',
 '(',   ')',   '*'    '+'       ',',   '-',   '.',   '/',
 '0',   '1',   '2',   '3',      '4',   '5',   '6',   '7',
 '8',   '9',   ':',   ';',      '<',   '='    '>',   '?',

 '@',   'A',   'B',   'C',      'D',   'E',   'F',   'G',
 'H',   'I',   'J',   'K',      'L',   'M',   'N',   'O',
 'P',   'Q',   'R',   'S',      'T',   'U',   'V',   'W',
 'X',   'Y',   'Z',   '[',      '\',   ']',   '¬',   '_',

 ' ',   'a',   'b',   'c',      'd',   'e',   'f',   'g',
 'h',   'i',   'j',   'k',      'l',   'm',   'n',   'o',
 'p',   'q',   'r',   's',      't',   'u',   'v',   'w',
 'x',   'y',   'z',   '{',      '|',   '}',   '~',   del);
```

INSTRUCTOR NOTES

SECTION 8
ARRAY TYPES AND ITERATIVE CONTROL STRUCTURES

INSTRUCTOR NOTES

SECTION 8
ARRAY TYPES AND ITERATIVE CONTROL STRUCTURES

COMPOSITE TYPES

ARRAY DECLARATIONS

ARRAY VALUES

ATTRIBUTES AND OPERATIONS

ARRAY OBJECTS - SPECIAL CASES

UNCONSTRAINED ARRAY TYPES

STRINGS

CONTROL STRUCTURES

INSTRUCTOR NOTES

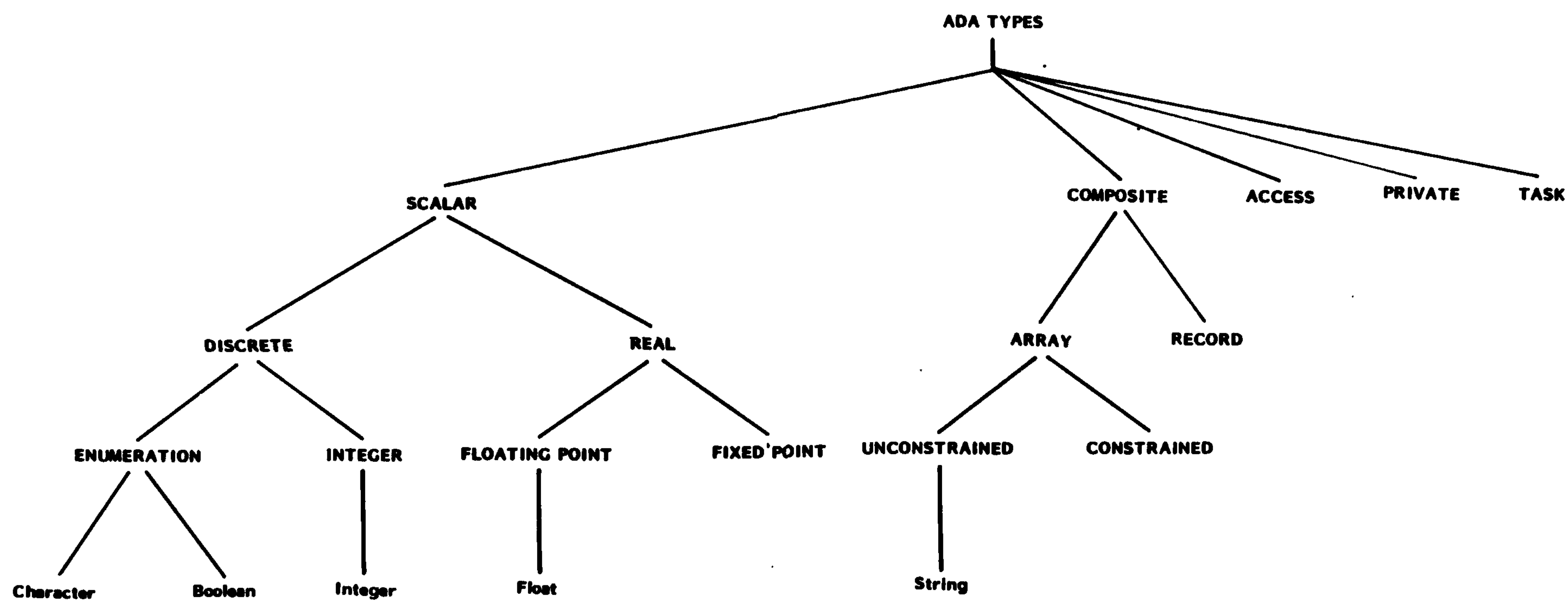
INDICATE THAT UP TO THIS POINT THE LEFT BRANCH OF THE TREE HAS BEEN PRESENTED.

NOW WE WILL EXAMINE COMPOSITE TYPES. ARRAYS WILL BE COVERED IN THIS SECTION AND RECORDS IN SECTION 9.

ACCESS TYPES WILL BE COVERED IN SECTION 10.

PRIVATE AND TASK TYPE WILL BE COVERED IN DETAIL IN L305 AND L401 RESPECTIVELY.

TYPE TREE


ADA TYPES
SCALAR
COMPOSITE
ACCESS
PRIVATE
TASK
DISCRETE
REAL
ARRAY
RECORD
ENUMERATION
INTEGER
FLOATING POINT
FIXED POINT
UNCONSTRAINED
CONSTRAINED
Character
Boolean
Integer
Float
String

INSTRUCTOR NOTES

WHAT IS AN ARRAY? IT IS A CLASS OF <u>COMPOSITE</u> TYPES.

COMPOSITE MEANS IT IS MADE (OR COMPOSED) OF SMALLER PIECES.

POINT OUT THAT SCALAR TYPES HAVE NO COMPONENTS WHEREAS COMPOSITE TYPES DO. SPECIFICALLY EACH OBJECT OF THE TYPE AT ANY TIME CONTAINS ONE VALUE.

MENTION THAT THE TYPES THEY HAVE BEEN STUDYING UP TO THIS POINT ARE SCALAR TYPES.

POINT OUT THAT WE MAY HAVE ARRAYS OF ARRAYS, ARRAYS OF RECORDS, AND RECORDS WHOSE COMPONENTS MAY BE EITHER ARRAYS OR RECORDS.

SCALAR VS. COMPOSITE

SCALAR TYPES

- CANNOT BE DECOMPOSED FURTHER
- BUILDING BLOCKS OF OTHER TYPES

EXAMPLES OF SCALAR TYPES:

NUMERIC TYPES

ENUMERATION TYPES

COMPOSITE TYPES

- CONTAIN COMPONENTS
- THE TYPE OF THE COMPONENTS MAY BE SCALAR OR COMPOSITE

EXAMPLES OF COMPOSITE TYPES:

ARRAYS - INDEXED COLLECTION OF SIMILAR (HOMOGENEOUS) COMPONENTS

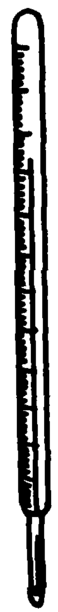
RECORDS - COLLECTION OF POTENTIALLY DIFFERENT (HETEROGENEOUS) COMPONENTS

INSTRUCTOR NOTES

THERE ARE OTHER ADVANTAGES OF ARRAYS ... WE WILL SEE HOW WE CAN "SLIDE" THE 29 MOST RECENT VALUES DOWN 1 POSITION IN THE ARRAY TO MAKE ROOM FOR THE NEWEST READING WHEN WE STUDY ARRAY OPERATIONS.

WHY ARRAYS?

SUPPOSE WE NEED TO KEEP TRACK OF THE 30 MOST RECENT READINGS OF A TEMPERATURE SENSOR.



USING SCALAR TYPES, WE ARE REQUIRED TO USE 30 UNIQUELY NAMED VARIABLES TO STORE THE VALUES BECAUSE EACH SCALAR OBJECT MAY CONTAIN ONLY ONE VALUE.

TEMPERATURE_1	TEMPERATURE_2	TEMPERATURE_3	...	TEMPERATURE_30
30.0	21.0	35.1		40.1

IT IS MUCH LESS CUMBERSOME TO STORE THESE VALUES IN ONE VARIABLE, STRUCTURED IN A WAY WHICH ALLOWS US TO USE ONE NAME TO REFER TO THE COLLECTION OF THESE VALUES.

	1	2	3		30
TEMPERATURES				...	

ARRAYS CAN BE USED TO GROUP VALUES OF IDENTICAL TYPE!

INSTRUCTOR NOTES

POINT OUT THAT THIS IS JUST AN EXAMPLE TO SHOW WHAT IT LOOKS LIKE - TO SATISFY CURIOSITY BEFORE SWINGING INTO THE SYNTAX AND RULES.

THE TYPE DECLARATION LOOKS A LITTLE DIFFERENT FROM OTHER TYPE DECLARATIONS WE'VE SEEN.

POINT OUT THE KEYWORD ARRAY.

THE FIRST PART OF THIS LECTURE WILL SHOW IN DETAIL HOW TO DECLARE ARRAY TYPES AND WHAT FLEXIBILITY THE USER HAS.

POINT OUT THAT THE OBJECT DECLARATION IS NO DIFFERENT FROM OTHER OBJECT DECLARATIONS. IT FOLLOWS THE GENERAL RULE OF OBJECT NAME, COLON, TYPE NAME.

ENSURE THAT THE STUDENTS UNDERSTAND THAT AN OBJECT DECLARED TO BE OF THE TYPE, Temperature_Type, IS A VARIABLE WHICH HAS 30 COMPONENTS, EACH OF WHICH IS OF TYPE FLOAT.

ARRAY TYPE EXAMPLE

TYPE DECLARATION:

```
type Temperature_Type is array (1 .. 30) of Float;
```

OBJECT DECLARATION:

```
Boiler_Room_Temperature : Temperature_Type;
```

1	2	3	4		28	29	30
37.6	34.2	43.5	30.1	...	29.9	41.0	36.3

INSTRUCTOR NOTES

POINT OUT THAT EACH OF THESE Component_Type, DIMENSIONS, AND Index_Subtypes WILL BE ADDRESSED IN THE FOLLOWING SLIDES.

POINT OUT THAT THE NUMBER OF INDICES IS DETERMINED BY THE NUMBER OF Index_Subtypes LISTED IN THE TYPE DECLARATION. ALSO, THERE IS NO LIMIT TO THE NUMBER OF INDICES ALLOWED.

SYNTAX OF AN ARRAY TYPE DECLARATION

SYNTAX:

```
type Type_Name is array (Index_Subtype {, Index_Subtype}) of Component_Type;
```

- Component_Type -- THE TYPE OF THE ELEMENTS OF THE ARRAY
- DIMENSION(S) -- THE NUMBER OF INDICES
- Index_Subtype -- THE SUBTYPE OF THE ARRAY INDEX IN A GIVEN POSITION (MAY BE A SUBTYPE OF ANY INTEGER OR ENUMERATION TYPE)

INSTRUCTOR NOTES

POINT OUT THAT 36.5 AND 43.19 ARE ONLY <u>POSSIBLE</u> VALUES FOR COMPONENTS OF Boiler_Room_Temperature. THE OBJECT Boiler_Room_Temperature, AS DECLARED HERE, HAS NO INITIAL VALUE.

DIMENSION ARROW POINTS TO FACT THAT THERE IS ONE COLUMN IN THIS GRAPHICAL REPRESENTATION OF THE ARRAY.

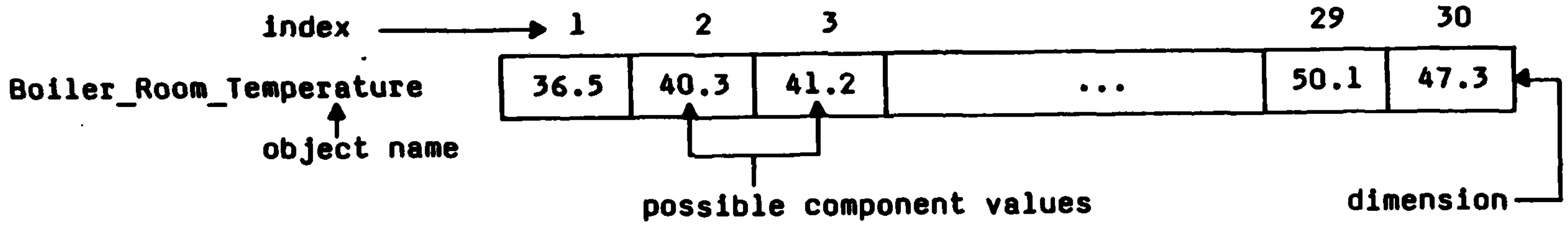
THE INDEX SUBTYPE TELLS US THAT THERE ARE 30 COMPONENTS AND THAT THEY ARE ACCESSED THROUGH AN INTEGER-VALUED INDEX. AN INDEX VALUE OF 0, 31, 121, ETC... IS ILLEGITIMATE ... AND CREATES A RUNTIME ERROR.

THE INSTRUCTOR SHOULD NOT GO INTO RUNTIME ERRORS IN DEPTH, THEY WILL BE COVERED IN THE SECTION ON EXCEPTIONS.

ARRAY TYPE EXAMPLE

```
                               Index_Subtype  Component_Type
type Temperature_Type is array (1 .. 30) of Float;
```

```
Boiler_Room_Temperature : Temperature_Type;
```



INSTRUCTOR NOTES

COMPONENT TYPE

CONTEXT:

```
type Temperature_Type is array (1 .. 30) of Float;
Boiler_Room_Temperature : Temperature_Type;
```

ILLEGAL

1	2.2	2	4.5	-3	-4	10.1	9.0		...		40
1	2	3	4	5	6	7	8				30

WHY? IT MIXES INTEGERS AND FLOATING POINT NUMBERS.

LEGAL

1.0	2.2	2.0	4.5	-3.0	-4.0	10.1	9.0		...		40.0
1	2	3	4	5	6	7	8				30

WHY? ALL OF THE COMPONENT VALUES ARE VALUES OF TYPE Float.

THE COMPONENTS MUST BE THE SAME (HOMOGENEOUS) TYPE.

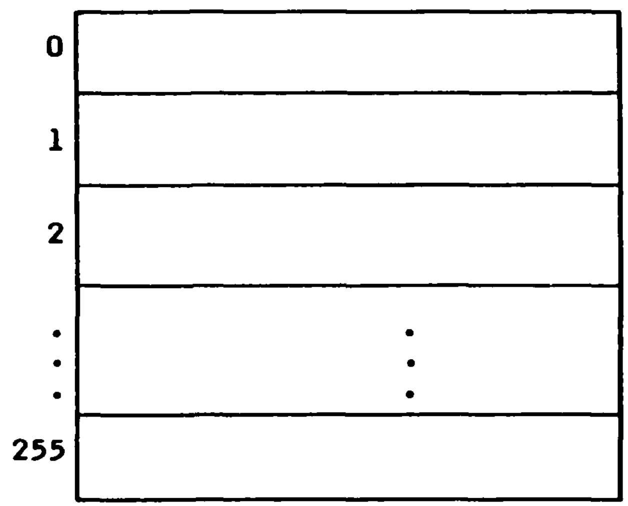
INSTRUCTOR NOTES

POINT OUT THAT THE REASON FOR WHY A COMPONENT MUST BE CONSTRAINED WILL BECOME CLEAR AFTER DISCUSSING UNCONSTRAINED ARRAYS. BASICALLY UNCONSTRAINED ARRAY TYPES ALLOW US TO DECLARE ARRAY TYPES OF UNSPECIFIED LENGTH. AN UNCONSTRAINED ARRAY CAN NOT BE USED AS A COMPONENT TYPE IN ANOTHER ARRAY TYPE DECLARATION.

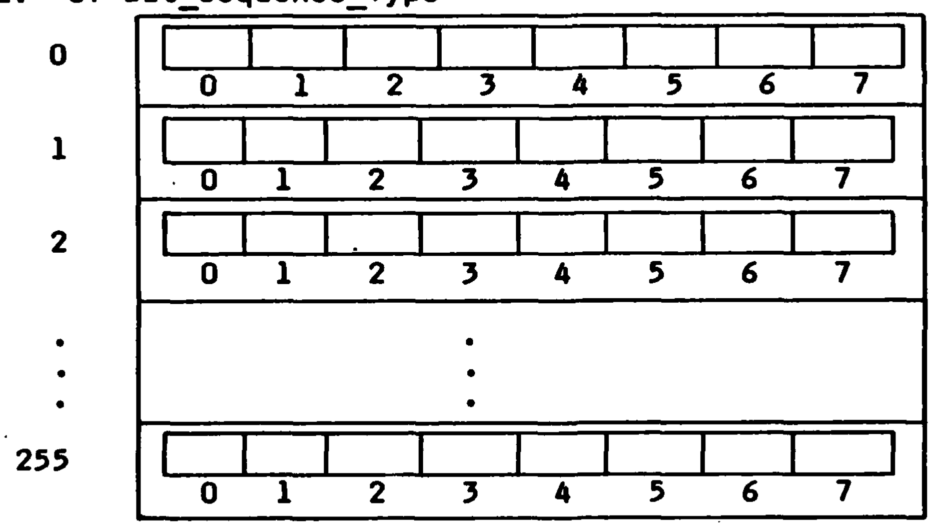
THE FIRST THREE EXAMPLES HAVE COMPONENTS OF TYPE Boolean, Integer, AND Float, RESPECTIVELY. THE FOURTH HAS AS ITS COMPONENT TYPE, THE TYPE Bit_Sequence_Type. POINT OUT THAT Ada ALLOWS ARRAYS OF ARRAYS.

POINT OUT THAT YOU CAN BE VERY CREATIVE WITH THE COMPONENT TYPE. DRAW A TEMPLATE FOR Memory_Type (USE TWO COLORS IF AVAILABLE).

1. array (0 .. 255)



2. of Bit_Sequence_Type



IF WE LOOK AT A SINGLE COMPONENT OF AN OBJECT Memory OF TYPE Memory_Type, WE ARE IN FACT LOOKING AT AN ARRAY CONTAINING 8 BOOLEAN COMPONENTS. MAKE SURE STUDENTS UNDERSTAND THAT A COMPONENT OF Memory IS <u>NOT</u> TRUE/FALSE BUT AN ARRAY OF TRUE/FALSE.

COMPONENT TYPE (Continued)

- TYPE OF THE VALUES STORED IN THE ARRAY

EXAMPLES:

ENUMERATION COMPONENT

```
type Bit_Sequence_Type is array (0 .. 7) of Boolean;
```

INTEGER COMPONENT

```
type Test_Scores_Type is array (1 .. 20) of Integer;
```

REAL COMPONENT

```
type Temperature_Type is array (1 .. 30) of Float;
```

COMPOSITE TYPE COMPONENT

```
type Memory_Type is array (0 .. 255) of Bit_Sequence_Type;
```

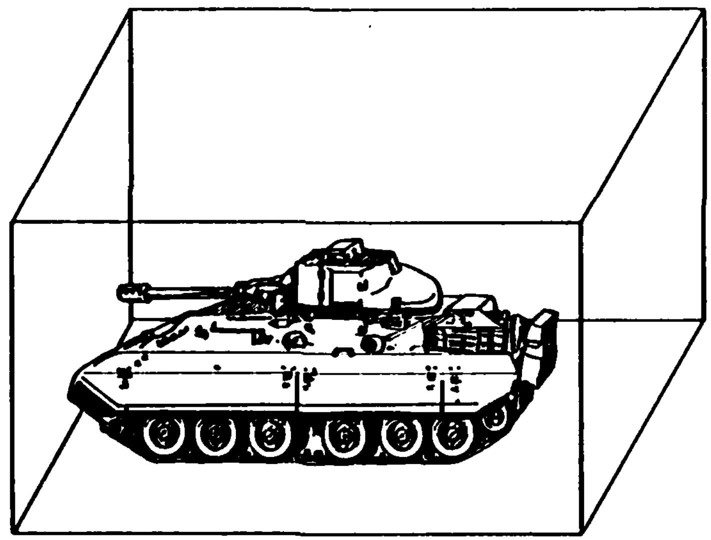
INSTRUCTOR NOTES

EXPLAIN AS THE TANK MOVES THERE IS A MATHEMATICALLY DEFINED TRANSFORMATION OF EACH POINT FROM ITS OLD POSITION TO ITS NEW POSITION. WE THINK ABOUT AND MANIPULATE THE TANK IN THREE-SPACE, AND THEREFORE THE MOST APPROPRIATE MODEL FOR REPRESENTING THE PHYSICAL SPACE IN A GIVEN COORDINATE SYSTEM IS A 3-DIMENSIONAL ARRAY. THIS IS DISTINCT FROM THE REPRESENTATION USED FOR THE PROJECTION OF THE TANK ONTO A 2-DIMENSIONAL GRAPHICS SCREEN. (AS THE TANK MOVES IN 3-DIMENSIONS, ITS 2-DIMENSIONAL PROJECTED IMAGE WOULD BE RECALCULATED AND REDISPLAYED.) FOR EXAMPLE, GUESSING FROM THE DRAWING, THE ARRAY CELL OR POINT (0, 0, 0) BEING A CORNER WOULD CONTAIN THE VALUE Not_Object SINCE THE TANK DOES NOT OCCUPY THAT POINT. HOWEVER, IF (100, 50, 0) AND (100, 50, 43) REPRESENTED POINTS ON THE TANK THEN THE Point_Characteristic_Type VALUE WOULD BE Object.

DIMENSION

CONSIDER THE PROBLEM OF REPRESENTING A 3 DIMENSIONAL OBJECT ON A GRAPHIC SCREEN.

EACH POINT IN 3-SPACE IS REPRESENTED BY A VALUE (Object, Not_Object) DEPENDING ON WHETHER THE POINT IS PART OF THE OBJECT (TANK) OR NOT (FREE SPACE).



```
type Point_Characteristic_Type is (Object, Not_Object);
type Coordinate_3D_System is array (0 .. 239, 0 .. 319, 0 .. 239)
                                    of Point_Characteristic_Type;
Tank_Position : Coordinate_3D_System;
```

INSTRUCTOR NOTES

DIMENSION CAN BE EASILY FIGURED OUT BY COUNTING HOW MANY Index_Subtypes EXIST.

THE CONTEXT FOR Color_Screen_Type AND Coordinate_3D_System ARE ON SLIDES 8-13 AND 8-10 RESPECTIVELY.

POINT OUT THAT THE TYPES Memory_Type AND Color_Screen_Type ARE ONE- AND TWO-DIMENSIONAL ARRAY TYPES RESPECTIVELY, EVEN THOUGH BOTH DEFINE AN ARRAY OF ARRAYS.

DIMENSION (Continued)

- ARRAYS MAY HAVE "N" DIMENSIONS

- DIMENSION IS INFERRED FROM THE SPECIFICATION OF THE INDICES

1-DIMENSIONAL

```
type Temperature_Type is array (1 .. 30) of Float;
type Bit_Sequence_Type is array (0 .. 7) of Boolean;
type Memory_Type is array (0 .. 255) of Bit_Sequence_Type;
```

2-DIMENSIONAL

```
type Color_Screen_Type is array (0 ..239, 0 .. 319) of Color_Type;
type Memory_Type_2 is array (0 .. 255, 0 .. 7) of Boolean;
```

3-DIMENSIONAL

```
type Coordinate_3D_System is array (0 .. 239, 0 .. 319, 0 .. 239)
                                    of Point_Characteristic_Type;
```

INSTRUCTOR NOTES

SO FAR, WE'VE ONLY SHOWN EXAMPLES WITH INTEGER TYPE INDICES WITH THE EXCEPTION OF THE COLOR GRAPHICS SCREEN.

POINT OUT THAT THESE ARE ALLOWABLE WAYS OF REPRESENTING AN INDEX.

ASK CLASS WHERE THEY HAVE ALREADY SEEN A REQUIREMENT FOR A DISCRETE TYPE. (CASE STATEMENT).

REFRESH STUDENTS ON WHAT SUBTYPES ARE.

FOR EXAMPLE:

```
subtype Color_Quantity_Type is Integer range 0 .. 15;
subtype Positive is Integer range 1 .. Integer'Last;
subtype Cruising_Altitude_Type is Altitude_Type range 15000 .. 35000;
subtype Word_Length is Line_Length range 1 .. 30;
subtype Characters_For_Line is Line_Length;
```

3. INDEX_SUBTYPE

- ALLOWS ONE TO IDENTIFY AND USE INDIVIDUAL COMPONENTS

- INDEX CAN BE ANY DISCRETE TYPE

  - ENUMERATION Color_Intensity_Type

RED	GREEN	BLUE
3	1	0

  - INTEGER Wait_Queue_Type

1	2	3	4	5
60	35	45	15	40

(MAY NOT BE REAL OR COMPOSITE)

INSTRUCTOR NOTES

POINT OUT THAT THE THREE FORMS FOR Index_Subtype ARE ALTERNATIVE WAYS OF EXPRESSING THE INDEX.

POINT OUT THAT FOR NOW EACH INDEX IS EXPRESSED IN ONE OF THESE FORMS, YET MULTIPLE INDICES MAY BE OF DIFFERENT FORMS. A FOURTH FORM (UNCONSTRAINED ARRAYS) IS DISCUSSED LATER.

POINT OUT THAT BECAUSE EVERY TYPE IS A SUBTYPE OF ITSELF, TYPE NAMES SUCH AS INTEGER MAY BE USED AS THE Index_Subtype.

3. INDEX_SUBTYPE (Continued)

- THREE FORMS FOR Index_Subtype

1. Lower_Bound .. Upper_Bound

```
type Temperature_Type is array (1 .. 30) of Float;
```

2. Subtype_Name

```
subtype Number_of_Acres is Integer range 1 .. 50;
type Yield_Type is array (Number_of_Acres) of Float;
   -- Indices will range from 1 to 50
type Color_Type is (Red, Green, Blue);
type Color_Intensity_Type is array (Color_Type) of
                                    Color_Quantity_Type;
   -- Color_Quantity_Type is an Integer Subtype ranging from 0 to 15
```

3. Subtype_Name range Lower_Bound .. Upper_Bound

```
type Yield_of_Front_Forty is array (Number_of_Acres range 1 .. 40)
                                    of Float;
```

INSTRUCTOR NOTES

WALK THROUGH EACH EXAMPLE POINTING OUT THE VARIOUS INDICES AND COMPONENTS.

AS A REVIEW QUESTION, ASK CLASS WHY Natural IS A BETTER CHOICE THAN Integer AS THE COMPONENT TYPE OF Number_Of_Coins_Type (IT PROVIDES A CONSTRAINT SO THAT ONE CANNOT HAVE A NEGATIVE NUMBER OF COINS).

IF THE TEACHER DEEMS IT NECESSARY, (S)HE MAY WISH TO DRAW BOXES TO REPRESENT THE ARRAY(S). BE SURE TO DECLARE AN OBJECT FOR THE TYPE BEFOREHAND.

EXAMPLES

CONTEXT:

```
type Coin_Type is (Penny, Nickel, Dime, Quarter, Half_Dollar);
```

EXAMPLES:

```
type Coin_Collection_Type is array (Coin_Type) of Natural;
type Parking_Meter_Coin_Table_Type is array
    (Coin_Type range Nickel .. Quarter) of Natural;
```

CONTEXT:

```
type Transmission_Mode_Type is (Asynchronous, Synchronous);
type Channel_Number_Type is range 0 .. 26;
```

EXAMPLE:

```
type Channel_Transmission_Mode_Type is array
    (Channel_Number_Type) of Transmission_Mode_Type;
```

INSTRUCTOR NOTES

DISCUSS DIMENSIONS VERSUS COMPONENTS.

A GRID MIGHT BE USED FOR A MINIATURE SPREAD SHEET. OR, IT COULD BE A CITY MAP WHOSE STREETS ARE NAMED A STREET, B STREET, ETC., AND WHOSE AVENUES ARE NUMBERED.

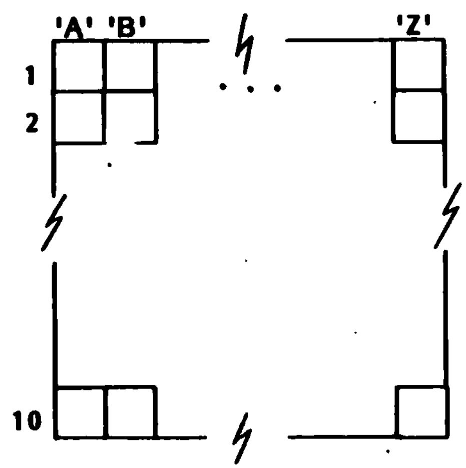
THE POINT OF THIS SLIDE IS TO SHOW THAT YOU MAY HAVE INDICES OF DIFFERENT TYPES.

MULTIDIMENSIONAL ARRAYS

- MAY HAVE INDICES OF DIFFERENT TYPES

```
type Grid_Type is array (Integer range 1 .. 10,
                         Character range 'A' .. 'Z') of Boolean;
```

- WILL ALLOW US TO REPRESENT A GRID



INSTRUCTOR NOTES

```
type Color_Component_Type   is (Red, Green, Blue);
subtype Color_Quantity_Type is Integer range 0 .. 15;
type Color_Type             is array (Color_Component_Type) of Color_Quantity_Type;
type Color_Screen_Type      is array (0 .. 239, 0 ..319) of Color_Type;
```

THIS ACTUALLY MODELS A RAMTEK RASTER SCREEN GRAPHICS DISPLAY SYSTEM GX-100B THE HARDWARE PROVIDES A 12 BIT WORD CORRESPONDING TO EACH PIXEL. THIS WORD IS DECODED INTO THE RED, GREEN, AND BLUE COMPONENTS. AN ALTERNATIVE IS:

```
type Color_Type is range 0 .. 4095;
```

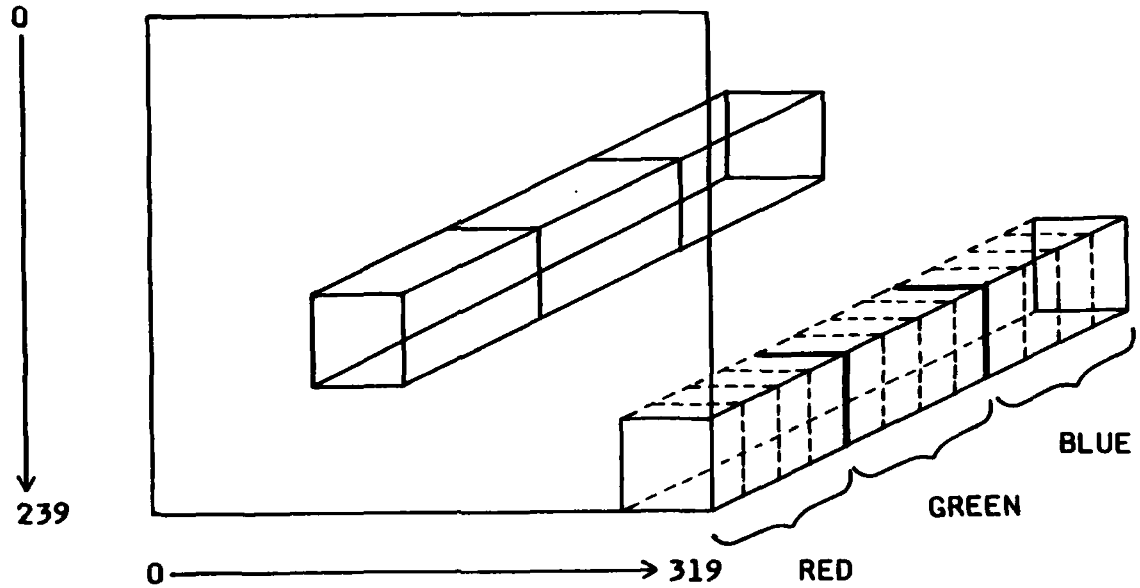
SINCE 12 BITS GIVES YOU 4096 POSSIBILITIES OF COLOR COMBINATIONS THIS IS A PLAUSIBLE REPRESENTATION. HOWEVER, THIS DOES NOT CORRESPOND TO THE RGB MODEL. THE RGB MODEL REPRESENTS COLOR VALUES IN TERMS OF THE QUANTITIES OF RED, GREEN, AND BLUE THAT MAKE UP THE DESIRED COLOR I.E., A COLOR TRIPLE FOR REPRESENTING A SHADE OF PURPLE MIGHT BE (15, 4, 11). THIS IS MORE MEANINGFUL THAN ITS EQUIVALENT NUMBER REPRESENTATION, 3915

$$\underset{\text{RED}}{\underbrace{1111}_{F}} \quad \underset{\text{GREEN}}{\underbrace{0100}_{4}} \quad \underset{\text{BLUE}}{\underbrace{1011}_{B}} \quad \Rightarrow 15*64 + 4*16 + 11 = 3915$$

EXERCISE

CONSIDER A COLOR GRAPHICS SCREEN WHICH IS 240 x 320 PIXELS. THE ACTUAL COLOR DISPLAYED IS DETERMINED BY HOW MUCH RED, GREEN, AND BLUE ARE "MIXED". FOR EXAMPLE, VALUES OF 15 (MAXIMUM VALUE POSSIBLE) FOR RED, GREEN, AND BLUE PRODUCE THE COLOR WHITE. THIS RESULTS IN A TOTAL OF 4096 COLORS THAT CAN BE DISPLAYED.

HOW DO YOU IMPLEMENT THIS COLOR GRAPHICS SCREEN?



INSTRUCTOR NOTES

DON'T SPEND ALOT OF TIME ON THE EXAMPLE. POINT OUT THAT THE INITIAL VALUE IS CALLED AN AGGREGATE, AND THAT AGGREGATES ARE THE TOPIC TO BE DISCUSSED NEXT.

ARRAY OBJECT DECLARATIONS

SYNTAX:

```
1.    Array_Type_Object     : Array_Type_Name [:=(Initial_Value)];
                                        or
2.    Array_Constant_Object : constant Array_Type_Name := (Initial_Value);
```

OBJECTS ARE EITHER

<u>VARIABLE</u> - THE VALUE OF ANY COMPONENTS CAN BE CHANGED (I.E., THROUGH ASSIGNMENT) SEE #1.

<u>CONSTANTS</u> - THE VALUE OF EACH COMPONENT IS FIXED FOR LIFE, CAN <u>NOT</u> BE CHANGED. SEE #2.

CONTEXT:

```
type Coin_Type is (Penny, Nickel, Dime, Quarter, Half_Dollar);
type Coin_Collection_Type is array (Coin_Type) of Natural;
```

EXAMPLE:

```
Coin_Values : constant Coin_Collection_Type := (1, 5, 10, 25, 50);
```

INSTRUCTOR NOTES

IF THEY WONDER WHY THE PICTURE IN THE EXAMPLE HAS VALUES TELL THEM THEY CAN ASSUME Boiler_Room_Temperature SOMEHOW WAS ASSIGNED TO.

NOTATION FOR ACCESSING COMPONENT OF AN ARRAY OBJECT

(INDIVIDUAL COMPONENT)

SYNTAX:

```
Array_Object (Index_Value {, Index_Value})
```

CONTEXT: Temperature_Type is array (1 .. 30) of Float;

Boiler_Room_Temperature : Temperature_Type;

EXAMPLE: Boiler_Room_Temperature (28) Value : 65.0

63.5	69.5	67.1	63.9		...		69.1	65.0	67.1	69.3
1	2	3	4				27	28	29	30

VG 728.2

INSTRUCTOR NOTES

THE EXAMPLES ON THIS PAGE ILLUSTRATE THE SYNTAX. A MORE MEANINGFUL EXAMPLE DEMONSTRATING THE USE OF SUBTYPES OCCURS ON THE NEXT SLIDE.

NOTATION FOR ACCESSING COMPONENTS OF AN ARRAY OBJECT (SLICE)

SYNTAX:

```
Array_Object (Starting_Position .. Ending_Position)
Array_Object (Subtype_Name)
```

A SLICE IS A PART OF A ONE-DIMENSIONAL ARRAY, CONSISTING OF CONSECUTIVE COMPONENTS.

EXAMPLE: Boiler_Room_Temperature (25 .. 27)   Value (67.2, 65.1, 65.0)

70.2	65.8	...	67.2	65.1	65.0	66.5	64.3	68.1
1	2		25	26	27	28	29	30

CONTEXT:

BUILDING D'S TEMPERATURE IS REPRESENTED BY THE ENTRIES IN 25, 26, AND 27

```
subtype Bldg_D_Entries is Integer range 25 .. 27;
```

EXAMPLE: Boiler_Room_Temperature (Bldg_D_Entries)   Value : (67.2, 65.1, 65.0)

INSTRUCTOR NOTES

THE MAIN MESSAGE HERE IS THAT IN Ada YOU CAN HAVE ARRAYS OF ANY COMPONENT TYPE. THE LANGUAGE POSES NO RESTRICTIONS!

POINT OUT THAT AN ARRAY OF ARRAYS IS THE LOGICAL WAY TO REPRESENT A LIST OF PHONE NUMBERS. A MATRIX OF NUMBERS WOULD BE INAPPROPRIATE HERE AND WOULD NOT CONVEY THE RIGHT INFORMATION.

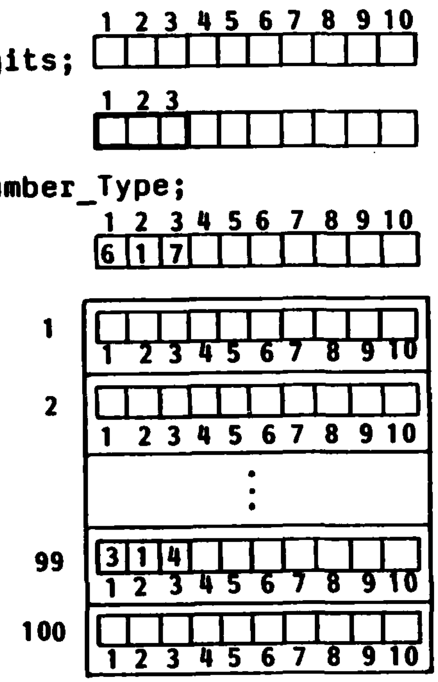
POINT OUT THAT THE USE OF THE Subtype_Name ENHANCES READABILITY. Area_Code IS MUCH MORE MEANINGFUL THAN 1 .. 3. SUPPOSE FOR SOME REASON, THE AREA CODE IS STORED AT THE END OF THE NUMBER. THE NUMERICAL SLICE 8 .. 10 WOULD NOT BE INHERENTLY OBVIOUS AS THE AREA CODE.

"314" IS THE ST. LOUIS AREA CODE.

EXAMPLE: ACCESSING COMPONENTS OF AN ARRAY OBJECT (ARRAY OF ARRAYS)

CONTEXT:

```
type Phone_Digits is range 0 .. 9;
type Telephone_Number_Type is array (1 .. 10) of Phone_Digits;
subtype Area_Code is Integer range 1 .. 3;
type Active_Phone_List is array (1 .. 100) of Telephone_Number_Type;
Phone_Number : Telephone_Number_Type;
Phone_List : Active_Phone_List;
```



EXAMPLES:

```
Phone_List (99) (Area_Code) := (3, 1, 4);
Phone_Number (Area_Code) := (6, 1, 7);
```

INSTRUCTOR NOTES

ANSWERS: False True

False

True

.
.
.

Penny Nickel Dime Quarter Half Dollar

.
.
.

False

3

(Nickel .. Quarter)

TO ACCESS COMPONENTS OF AN ARRAY OBJECT WHICH HAS AN ENUMERATION TYPE INDEX, USE THE ENUMERATION LITERAL OR AN OBJECT OF THE ENUMERATION TYPE.

DO NOT GET BOGGED DOWN IN A DISCUSSION OF AGGREGATES. THEY ARE THE NEXT TOPIC TO BE DISCUSSED.

STRESS THAT WITH 2-DIMENSIONAL ARRAYS, 2 Index_Subtypes MUST BE SPECIFIED.

ASK STUDENTS WHETHER And_Truth_Table (False .. True, False) IS LEGAL. (NO; YOU CAN ONLY TAKE SLICES OF A ONE-DIMENSIONAL ARRAY.)

EXERCISE

FILL IN INDEX VALUES AND OBJECT VALUES WHERE BLANK:

CONTEXT:

```
type Logical_Result_Type is array (Boolean, Boolean) of Boolean;
And_Truth_Table : Logical_Result_Type;
```

	____	____
____	False	False
____	False	True

```
type Coin_Type is (Penny, Nickel, Dime, Quarter, Half_Dollar);
type Coin_Collection_Type is array (Coin_Type) of Natural;
Coin_Count : Coin_Collection_Type;
```

____	____	____	____	____
3	5	4	6	2

EXAMPLES:	Value:
And_Truth_Table (False, True)	________
Coin_Count (Penny)	________
Coin_Count ( )	(5, 4, 6)

INSTRUCTOR NOTES

POINT OUT THAT STUDENTS WILL SEE AGGREGATES AGAIN WITH RECORDS.

AGGREGATES

- GROUP VALUES INTO A SINGLE ELEMENT
- ARE USED TO ASSIGN VALUES TO A COMPOSITE OBJECT AS A GROUP

INSTRUCTOR NOTES

NOW THAT THE STUDENTS HAVE A CLEAR UNDERSTANDING OF HOW TO ACCESS COMPONENTS OF ARRAY OBJECTS, WE CAN LOOK AT HOW ARRAYS ARE INITIALIZED. TAKE THE EXAMPLE WE HAVE SEEN AND POINT OUT THE AGGREGATES.

POINT OUT THAT NAMED (WITH DISCRETE RANGE) IS CLEARLY THE BEST REPRESENTATION OF THE VALUES FOR THIS EXAMPLE.

REMEMBER THE TELEPHONE NUMBER EXAMPLE ..

```
type Phone_Digits is range 0 .. 9;
type Telephone_Number_Type is array (1 .. 10) of Phone_Digits;
```

BRIEFLY EXPLAIN WHY FOR TELEPHONE NUMBERS, POSITIONAL NOTATION IS THE BEST REPRESENTATION BECAUSE IT MOST CLOSELY MODELS HOW WE THINK OF TELEPHONE NUMBERS, I.E., AS ONE CONTINUOUS ROW OF DIGITS.

POINT OUT THAT => IS A SINGLE LEXICAL SYMBOL.

NOTE: POSITIONAL AND NAMED MAY BE MIXED IN RECORD AGGREGATES.

AGGREGATES

ARRAY INITIALIZATION

SYNTAX:

```
POSITIONAL

        aggregate := (Component_Type_Value {, Component_Type_Value});

NAMED

        aggregate := (Index_Subtype_Values => Component_Type_Value {,
                      Index_Subtype_Values => Component_Type_Value});
```

CONTEXT:

```
type Coin_Type is (Penny, Nickel, Dime, Quarter, Half_Dollar);
type Coin_Collection_Type is array (Coin_Type) of Natural;
```

EXAMPLES:

```
POSITIONAL
        Coin_Count : Coin_Collection_Type := (13, 10, 6, 6, 6);

NAMED (WITH DISCRETE RANGE)
        Coin_Count : Coin_Collection_Type := (Penny => 13, Nickel => 10,
                           Dime => 6, Quarter => 6, Half_Dollar => 6);

NAMED (WITH BAR)
        Coin_Count : Coin_Collection_Type := (Penny => 13, Nickel => 10,
                           Dime | Quarter | Half_Dollar => 6);
```

NOTE: POSITIONAL AND NAMED MAY <u>NOT</u> BE MIXED IN <u>ARRAY</u> AGGREGATES!

INSTRUCTOR NOTES

A WAY OF REMEMBERING THAT AN ARRAY AGGREGATE MUST BE ALL POSITIONAL OR ALL NAMED IS THAT HOMOGENEOUS COMPONENTS ARE WRITTEN IN HOMOGENEOUS NOTATION.

AGGREGATES
POSITIONAL VS. NAMED NOTATION

POSITIONAL

- ORDER IS SIGNIFICANT

  CONTEXT:

```
type Phone_Digits is range 0 .. 9;
type Telephone_Number_Type is array (1 .. 10) of Phone_Digits;
Phone_Number : Telephone_Number_Type;
```

  EXAMPLE:

```
Phone_Number := (8, 0, 0, 5, 5, 5, 1, 0, 0, 0);
```

NAMED

- ORDER IS INSIGNIFICANT

  CONTEXT:

```
type Coin_Type is (Penny, Nickel, Dime, Quarter, Half_Dollar);
type Coin_Collection_Type is array (Coin_Type) of Natural;
Coin_Count : Coin_Collection_Type;
```

  EXAMPLE:

```
Coin_Count := (Quarter => 6, Nickel => 10, Penny => 13, Half_Dollar => 6,
                Dime => 6);
```

INSTRUCTOR NOTES

POINT OUT THAT "others" CAN BE USED TO INITIALIZE AN ENTIRE ARRAY.

DO NOT DISCUSS BULLET 3 EXTENSIVELY. SECTION 4.8 OF THE REFERENCE MANUAL DISCUSSES WHEN QUALIFICATION IS NOT REQUIRED (ACTUAL PARAMETERS).

FOR INSTRUCTORS INFORMATION:

GIVEN A : constant String := (2 => 'a', others => 'v');

THE AGGREGATE IS AMBIGUOUS AND LEADS TO THE FOLLOWING INTERPRETATIONS:

1	2	3	...	n
V	A	V		V

2	3	4	...	n
A	V	V		V

THE REFERENCE MANUAL REQUIRES THAT THE BOUNDS OF THE AGGREGATE BE COMPLETELY DETERMINED FROM ITS CONTEXT, WHICH DOES NOT INCLUDE ANY BOUNDS INFORMATION PRECEDING THE ":=" (I.E., String (1 .. 3) WOULD NOT HELP IN THE ABOVE CASE.)

WE NEED TO QUALIFY THIS

```
subtype String_1_3 is String (1 .. 3);
A : constant string := String_1_3'(2=> 'a', others => 'v');
```

AGGREGATES
"OTHERS"

SYNTAX:

```
POSITIONAL
  aggregate := ({Component_Type_Value} {, Component_Type_Value}, others =>
                 Component_Type_Value);
NAMED
  aggregate := array_type_name'(Index_Subtype_Values =>
                  Component_Type_Value{, Index_Subtype_Values =>
                  Component_Type_Value}, others => Component_Type_Value);
  aggregate := (Index_Subtype_Values => Component_Type_Value{,
                  Index_Subtype_Values => Component_Type Value}, others =>
                  Component_Type_Value);
```

- THE KEY WORD <u>others</u> MAY BE USED IN AGGREGATES AS THE LAST VALUE ASSOCIATION IN THE AGGREGATE TO ASSIGN THE SAME VALUE TO ALL COMPONENTS NOT EXPLICITLY ASSIGNED TO PREVIOUSLY.

- IT CAN BE USED IN AGGREGATES REPRESENTED IN EITHER POSITIONAL OR NAMED NOTATION.

- AGGREGATES WRITTEN IN NAMED NOTATION USING "others" MUST USUALLY APPEAR IN QUALIFIED EXPRESSIONS

  CONTEXT:

```
type Coin_Type is (Penny, Nickel, Dime, Quarter, Half_Dollar);
type Coin_Collection_Type is array (Coin_Type) of Natural;
Coin_Count : Coin_Collection_Type;
```

  EXAMPLE:

```
Coin_Count := Coin_Collection_Type'(Penny => 13, Nickel => 10,
                  others => 6);
```

INSTRUCTOR NOTES

"THE AGGREGATE MUST BE COMPLETE" MEANS THAT IT MUST CONTAIN A VALUE FOR <u>EVERY</u> COMPONENT IN THE OBJECT BEING ASSIGNED TO. STRESS THIS CONCEPT.

RULES FOR AGGREGATES

- AGGREGATE MUST BE COMPLETE

- EACH COMPONENT MUST BE GIVEN ONLY ONE VALUE

INSTRUCTOR NOTES

NAMED NOTATION IS REQUIRED IN ORDER TO AVOID CONFUSION WITH AN EXPRESSION ENCLOSED IN PARENTHESES.

AGGREGATES
SPECIAL CASE

- AN AGGREGATE CONTAINING A ONE COMPONENT ASSOCIATION MUST BE REPRESENTED IN NAMED NOTATION.

CONTEXT:

```
type One_Component_Array_Type is array (Integer range 1 .. 1) of Float;
One_Component_Array : One_Component_Array_Type;
```

EXAMPLE:

```
One_Component_Array := (4.5);          -- **ILLEGAL
One_Component_Array := (1 => 4.5);     -- LEGAL AND MANDATORY
```

INSTRUCTOR NOTES

THE PURPOSE OF THIS SLIDE IS TO SHOW HOW MULTIDIMENSIONAL ARRAY AGGREGATES ARE WRITTEN. THE *FIRST* EXAMPLE USES ALL POSITIONAL NOTATION. THE SECOND USES ALL NAMED NOTATION. THE THIRD IS A MIXING OF THE TWO NOTATIONS. POINT OUT THAT MIXING THE NOTATION IS NOT ALLOWED (EXCEPT FOR <u>others</u>) WITHIN A SINGLE ARRAY AGGREGATE. THE MIXING IS ALLOWED FOR SUBAGGREGATES. (POINT OUT THE SUBAGGREGATES.) FOR EXAMPLE,

```
Add_Truth_Table : constant Logical_Result_Type := ((False, True), True => (False,
                                                                          True));
```

IS ILLEGAL.

MULTIDIMENSIONAL ARRAY AGGREGATE

CONTEXT:

```
type Logical_Result_Type is array (Boolean, Boolean) of Boolean;
```

EXAMPLE:

1. XOR

	False	True
False	False	True
True	True	False

```
XOR_Truth_Table : constant Logical_Result_Type := ((False, True), (True, False));
```

2. OR

	False	True
False	False	True
True	True	True

```
OR_Truth_Table : constant Logical_Result_Type := (False => (False => False,
                          True => True) True => (False => True, True => True));
```

3. AND

	False	True
False	False	False
True	False	True

```
AND_Truth_Table : constant Logical_Result_Type := (False => (False => False,
                                     True => False), True => (False, True));
```

INSTRUCTOR NOTES

ANSWERS:

1. A.
```
Control_Word_1 : Bit_Sequence_Type := (False, True, True, False, True, False,
                                       True, False);
```
   B.
```
Control_Word_2 : Bit_Sequence_Type := (False, True, False, others => True);
```
   C.
```
Control_Word_3 : Bit_Sequence_Type := (others => False);
```

AGGREGATES EXERCISE

CONTEXT: type Bit_Sequence_Type is array (0 .. 7) of Boolean;

1. WRITE THE MOST APPROPRIATE AGGREGATE ASSIGNMENT FOR THE FOLLOWING:

A. Control_Word_1 : Bit_Sequence_Type :=

0	1	2	3	4	5	6	7
False	True	True	False	True	False	True	False

ANSWER: ______________________________

B. Control_Word_2 : Bit_Sequence_Type :=

0	1	2	3	4	5	6	7
False	True	False	True	True	True	True	True

ANSWER: ______________________________

C. Control_Word_3 : Bit_Sequence_Type :=

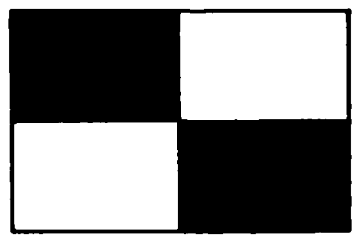
0	1	2	3	4	5	6	7
False	False	False	False	False	False	False	False

ANSWER: ______________________________

ANSWERS:

2. Flag : Color_Screen_Type := (0 .. 239 => (0 .. 159 => (15, 15, 15),
                                              others => (0, 0, 0)));

INSTRUCTOR MAY WISH TO EXPAND ON THIS AND DECLARE



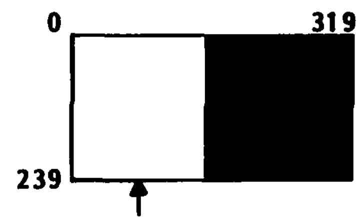
```
Flag_2 : Color_Screen_Type := (0 .. 119 => (0 .. 159 => (15, 15, 15),
                                            160 .. 319 =>(0, 0, 0)),
                               120 .. 239 => (0 .. 159 => (0, 0, 0),
                                            160 .. 319 => (15, 15, 15)));
```

AGGREGATES EXERCISE (Continued)

2. REMEMBER THE GRAPHICS SCREEN EXAMPLE. NOW INITIALIZE THE SCREEN SO THAT THE LEFT HALF IS WHITE (15, 15, 15) AND THE RIGHT HALF IS BLACK (0, 0, 0).

CONTEXT:

```
type Color_Component_Type is (Red, Green, Blue);

subtype Color_Quantity_Type is Integer range 0 .. 15;

type Color_Type is array (Color_Component_Type)
      of Color_Quantity_Type;

type Color_Screen_Type is array (0 .. 239, 0 .. 319)
      of Color_Type;
```



ANSWER: FLAG: ______________________ := ____

INSTRUCTOR NOTES

ANSWERS:	240	0 .. 319
	239	239
	240	240
	0 .. 239	0 .. 239

POINT OUT THAT THE NUMBER IN PARENTHESIS INDICATES WHICH INDEX TO APPLY THE ATTRIBUTE TO. *IT IS OPTIONAL.* WHEN *NONE IS SUPPLIED THE ATTRIBUTE* IS APPLIED TO THE FIRST INDEX.

A POSSIBLE CONVENTION (NOT ILLUSTRATED ON THIS SLIDE) IS TO ALWAYS USE "(N)" AS PART OF THE ATTRIBUTE *FOR MULTIDIMENSIONAL* ARRAYS (EVEN WHEN *N* = 1), AND *TO* ALWAYS *OMIT* IT FOR ONE-DIMENSIONAL ARRAYS.

HAVE STUDENTS FILL IN BLANKS.

WRITE ON SLIDE THAT WE MAY SIMILARLY DO ...

```
Coordinate_3D_System'First(1) _0_
```

ARRAY TYPE ATTRIBUTES

WHERE A IS EITHER AN ARRAY TYPE OR AN ARRAY OBJECT AND N IS A SPECIFIC DIMENSION.

```
A'First        A'First(N)
A'Last         A'Last(N)
A'Length       A'Length(N)
A'Range        A'Range(N)
```

CONTEXT:

```
type Point_Characteristic_Type is (Object, Not_Object);

type Coordinate_3D_System is array (0 .. 239, 0 .. 319, 0 .. 239)
                    of Point_Characteristic_Type;

Tank_Position : Coordinate_3D_System;
```

EXAMPLES:

Tank_Position'First	0	Tank_Position'First(2)	0
Tank_Position'Last	239	Tank_Position'Last(2)	319
Tank_Position'Length		Tank_Position'Length(2)	320
Tank_Position'Range	0 .. 239	Tank_Position'Range(2)	
Tank_Position'First(1)	0	Tank_Position'First(3)	0
Tank_Position'Last(1)		Tank_Position'Last(3)	
Tank_Position'Length(1)		Tank_Position'Length(3)	
Tank_Position'Range(1)		Tank_Position'Range(3)	

INSTRUCTOR NOTES

THIS SLIDE SUMMARIZES THE OPERATIONS. THEY ARE COVERED IN DETAIL ON SUBSEQUENT SLIDES.

CLARIFY "EQUAL-LENGTH ONE-DIMENSIONAL." IT MEANS THEY HAVE THE SAME RELATIVE INDICES, NOT THE SAME ABSOLUTE INDICES. (SAME NUMBER OF COMPONENTS.)

CATENATION: AN ARRAY TYPE CAN BE CATENATED TO ANY OTHER ARRAY TYPE THAT HAS THE SAME COMPONENT TYPE AND THE SAME INDEX TYPE.

WE HAVE JUST SEEN USE OF THE ASSIGNMENT OPERATOR WITH AGGREGATES.

ARRAY TYPE OPERATIONS

- ASSIGNMENT: :=
- EQUALITY/INEQUALITY: = /=
- CATENATION: & (FOR ONE-DIMENSIONAL ARRAYS ONLY)
- RELATIONAL: < <= > >= (FOR ONE-DIMENSIONAL ARRAYS WITH DISCRETE COMPONENTS ONLY)
- LOGICAL: not xor and or (FOR EQUAL LENGTH ONE-DIMENSIONAL ARRAYS WITH Boolean COMPONENTS ONLY)

INSTRUCTOR NOTES

IN ASSIGNMENTS, THE TYPE OF THE VALUE BEING ASSIGNED MUST BE THE SAME AS THE COMPONENT TYPE OF THE OBJECT BEING ASSIGNED TO.

ARRAY TYPE OPERATIONS -- ASSIGNMENT

- ASSIGNMENT OF COMPONENTS
- ASSIGNMENT OF AN ENTIRE ARRAY

CONTEXT FOR EXAMPLES:

```
type Test_Scores_Type is array (1 .. 20) of Integer;
type Vector_Type is array (1 .. 30) of Float;
type Coin_Type is (Penny, Nickel, Dime, Quarter, Half_Dollar);
type Monetary_Value_Type is array (Coin_Type) of Integer;
Old_Score, New Score : Test_Scores_Type;
Vector_1, Vector_2   : Vector_Type;
Monetary_Value       : Monetary_Value_Type;
```

EXAMPLES:

```
Old_Score := New_Score; -- ARRAY ASSIGNMENT
Vector_1(3) := Vector_2(1);
Monetary_Value (Nickel) := 5 * Monetary_Value (Penny);
```

INSTRUCTOR NOTES

POINT OUT THE USE OF SLICES. THE EXAMPLE IS ACTUALLY COMPARING TWO PARTS (SLICES) OF THE SAME ARRAY OBJECT.

THE ASSIGNMENT STATEMENT FOR Same_Scores ILLUSTRATES TWO POINTS.

1. EQUALITY COMPARISON BETWEEN TWO ARRAYS

2. ASSIGNMENT OF A BOOLEAN-VALUED EXPRESSION TO A BOOLEAN VARIABLE

ARRAY TYPE OPERATIONS -- EQUALITY/INEQUALITY

- RESULT IS Boolean

CONTEXT FOR EXAMPLES:

```
type Vector_Type is array (1 .. 30) of Float;
type Test_Scores_Type is array (1 .. 20) of Integer;
Vector_1, Vector_2   : Vector_Type;
Old_Score, New Score : Test_Scores_Type;
Same_Scores          : Boolean;
```

EXAMPLES:

```
if Vector_1 (15 .. 20) /= Vector_1 (21 .. 26) then
      .
      .
      .
end if;
if Vector_1(1) = Vector_2(1) then
      ...
end if;
Same_Scores :=  Old_Score = New_Score;
```

INSTRUCTOR NOTES

POINT OUT THAT THIS IS AN OPERATOR THAT THEY MAY NOT BE FAMILIAR WITH.

MENTION THAT WHEN USED ON ARRAYS, BOTH OPERANDS MUST HAVE THE SAME TYPE WHEN USED TO CATENATE AN ARRAY AND A SCALAR VALUE THE SCALAR VALUES MUST HAVE THE SAME TYPE AS THE ARRAYS' COMPONENT TYPE.

ARRAY TYPE OPERATIONS -- CATENATION

- CATENATION (&) JOINS TWO ONE-DIMENSIONAL ARRAYS TO FORM A NEW ARRAY.
- CAN ALSO JOIN A COMPONENT AND AN ARRAY, OR TWO COMPONENTS.
- USED PRIMARILY WITH STRINGS AND CONTROL CHARACTERS

CONTEXT FOR EXAMPLES:

```
type Bit_Sequence_Type is array (0 .. 7) of Boolean;
type Memory_Type is array (0 .. 255) of Bit_Sequence_Type;
Processor_Memory, Peripheral_Memory : Memory_Type;
type Test_Scores_Type is array (1 .. 20) of Integer;
Most_Recent_Scores : Test_Scores_Type;
New_Score          : Integer;
```

EXAMPLES:

```
Most_Recent_Scores := Most_Recent_Scores (1 .. 19) & New_Score;
Processor_Memory (0 .. 127) := Peripheral_Memory (192 .. 255) &
                               Peripheral_Memory (128 .. 191);
```

INSTRUCTOR NOTES

ARRAY TYPE OPERATIONS - RELATIONAL

>	>=
<	<=

- THESE ORDERING OPERATORS ARE PREDEFINED FOR ARRAYS WHICH

  1. ARE ONE-DIMENSIONAL
  2. HAVE COMPONENTS OF A DISCRETE TYPE

INSTRUCTOR NOTES

COMPARISON OF TWO ARRAYS PRODUCES A SINGLE BOOLEAN RESULT.

REMIND STUDENTS OF THE PREDEFINED ENUMERATION TYPE DECLARATION FOR Boolean PRESENTED IN CHAPTER 5.

```
type Boolean is (False, True);
```

ARRAY TYPE OPERATIONS - RELATIONAL

- ASSUME A AND B ARE ARRAYS WHICH CAN BE COMPARED AND A /= B.

  TO DETERMINE WHETHER A < B, THE Ada LANGUAGE:

  - FINDS THE <u>FIRST</u> POSITION AT WHICH A AND B DIFFER
  - COMPARES THE COMPONENTS AT THAT POSITION
  - IF THE COMPONENT OF A IS LESS THAN THE COMPONENT OF B THEN A < B.

CONTEXT:

```
type Bit_Sequence_Type is array (0 .. 7) of Boolean;
Control_Word          : Bit_Sequence_Type := (False, False, True, True,
                                              False, True, False, True);
Word_On_Parallel_Bus : Bit_Sequence_Type := (False, False, True, True,
                                              False, False, True, True);
```

EXAMPLE:

```
Control_Word > Word_On_Parallel_Bus yields True
```

BECAUSE THE COMPONENTS IN THE FIFTH POSITION (COUNTING FROM 0) DIFFER, AND THE VALUE TRUE IS GREATER THAN THE VALUE FALSE (BECAUSE OF THE ORDERING OF THESE LITERALS IN THE DECLARATION OF TYPE BOOLEAN).

INSTRUCTOR NOTES

POINT OUT ARRAYS SHOULD BE OF THE SAME LENGTH.

ARRAYS OF UNEQUAL LENGTH WILL COMPILE BUT NOT EXECUTE.

A CHECK IS MADE TO ENSURE THAT FOR EACH COMPONENT OF THE LEFT OPERAND THERE IS A MATCHING COMPONENT OF THE RIGHT OPERAND AND VICE VERSA. IT IS A RUNTIME ERROR IF THIS CHECK FAILS.

ARRAY TYPE OPERATIONS - LOGICAL

- THE LOGICAL OPERATIONS and, or AND xor ARE APPLIED ON A COMPONENT-BY-COMPONENT BASIS TO ONE-DIMENSIONAL ARRAYS OF Boolean VALUES, PRODUCING A NEW ONE-DIMENSIONAL ARRAY OF Boolean VALUES.

CONTEXT:

```
type Bit_Sequence_Type is array (0 .. 7) of Boolean;
A, B, C : Bit_Sequence_Type;
```

EXAMPLE:

	0	1	2	3	4	5	6	7
A:	True	True	True	True	False	False	False	False

```
A := (0 ..3 => True, 4 .. 7 => False);
```

	0	1	2	3	4	5	6	7
B:	True	True	False	False	True	True	False	False

```
B := (0|1|4|5 => True, 2|3|6|7 => False);
```

```
C := A xor B;
```

	0	1	2	3	4	5	6	7
now C equals	False	False	True	True	True	True	False	False

```
-- C = (0|1|6|7 => False, 2 .. 5 => True)
```

INSTRUCTOR NOTES

- FOR INTERESTED STUDENTS

  - THIS FEATURE IS NOT PROVIDED IN MANY OTHER LANGUAGES

  - DYNAMIC ARRAYS ARE MORE TYPICALLY USED IN BLOCKS WHERE THE SIZE OF THE NEEDED ARRAY IS DETERMINED (FROM PROMPTING THE USER OR FROM READING A FILE) AND THEN THIS SIZE IS USED IN THE ARRAY DECLARATION.

  - THE ACTUAL BOUNDS OF THE ARRAY CAN BE NAMED USING THE ATTRIBUTES DISCUSSED ON SLIDE 8-30.

DYNAMIC ARRAYS

IN A GIVEN DAY THERE ARE A VARIABLE NUMBER OF HELICOPTERS SERVICED. THE FOLLOWING PROCEDURE WILL DECLARE AN ARRAY OF THE APPROPRIATE SIZE FOR EACH CASE.

```
procedure Tabulate_Daily_Service_Hours (Number_of_Helicopters : in Positive) is

      type Man_Hours_Servicing_Choppers_Type is array (1 .. Number_of_Helicopters)
                                                        of Natural;
      Man_Hours_Servicing_Choppers : Man_Hours_Servicing_Choppers_Type;

begin

  --  input man hours spent on each of the helicopters

  --  serviced for the day

  --  do calculations, statistics

  --  run report

end Tabulate_Daily_Service_Hours;
```

INSTRUCTOR NOTES

THE PRINCIPLE EXPRESSED IN BULLET 2 WILL BE EXPANDED ON IN THE DISCUSSION OF UNCONSTRAINED ARRAYS. POINT BACK TO PREVIOUS SLIDE AS AN EXAMPLE OF THIS POINT. THE PROGRAM TREATS THE ARRAY UNIFORMLY NO MATTER HOW MANY COMPONENTS THE ARRAY HAS DURING ANY PARTICULAR EXECUTION.

DYNAMIC ARRAYS
NOTABLE POINTS

- ACTUAL NUMBER OF COMPONENTS IS DETERMINED AT RUNTIME (ELABORATION)

- THIS ABILITY ALLOWS PROGRAMS TO DEAL UNIFORMLY WITH ARRAYS, REGARDLESS OF THE NUMBER OF COMPONENTS OF THE ARRAY

INSTRUCTOR NOTES

- IT IS GENERALLY CONSIDERED POOR PRACTICE TO USE ANONYMOUS ARRAYS FOR ANY OTHER PURPOSE BECAUSE THE ABSENCE OF A TYPE NAME SEVERELY LIMITS WHAT CAN BE DONE WITH THE OBJECT. FOR EXAMPLE, IT CANNOT BE PASSED AS AN ACTUAL PARAMETER BECAUSE ITS TYPE WILL NOT MATCH THE TYPE OF THE FORMAL PARAMETER.

- NOTICE Octal_Range IS 0 .. 7 AND YET WE ONLY ASSIGNED 0 THROUGH 6. THERE IS NO OP-CODE CORRESPONDING TO 7 (DESIGNER'S DISCRETION).

- FOR ANOTHER EXAMPLE, YOU ARE DOING SOME PROCESSING ON CHEMICAL EXPERIMENTS AND NEED TO REFERENCE THE MELTING POINTS OF CERTAIN SOLIDS. CREATE A LOOKUP TABLE, INDEXED BY THE COMPOUND NAME OF MELTING POINTS.

HISTORICAL NOTE: ANONYMOUS TYPES ARE A HOLD OVER FROM AN EARLIER DESIGN OF THE LANGUAGE.

WHY ANONYMOUS ARRAY TYPES?

- RECOMMENDED FOR TABLE LOOK-UP APPLICATIONS ONLY

EXAMPLE:

```
subtype Octal_Range is 0 .. 7;
type Op_Codes_Type is (NOP, ADD, SUB, MULT, DIV, JMP, JNZ);
Octal_Op_Code_Table : constant array (Op_Codes_Type) of Octal_Range :=
                            (NOP  => 0,
                             ADD  => 1,
                             SUB  => 2,
                             MULT => 3,
                             DIV  => 4,
                             JMP  => 5,
                             JNZ  => 6);
```

INSTRUCTOR NOTES

- THE EXAMPLE FROM THE PREVIOUS SLIDE MAY ALSO BE USED TO POINT OUT THE ADVANTAGE OF USING A TYPE NAME. IT ENABLES US TO ASSIGN WHOLE ARRAYS THAT HAVE BEEN DECLARED SEPARATELY. IF WE HAD DECLARED

```
type Octal_Op_Code_Type is array (Op_Code_Type) of Octal_Range;
A, B : Octal_Op_Code_Type;
```

then A := B IS LEGAL AND MUCH MORE CONVENIENT THAN

A(1) := B(1), A(2) := B(2) ...

ANONYMOUS ARRAY OBJECT DECLARATIONS

SYNTAX:

```
Array_Type_Object     : array (Index_Subtype {, Index_Subtype } ) of
                        Component_Type_Name [:= (Initial_Value)];
Array_Constant_Object : constant array (Index_Subtype {, Index_Subtype } ) of
                        Component_Type_Name : = (Initial_Value);
```

- AN Initial_Value IS MANDATORY FOR CONSTANT ARRAY DECLARATIONS
- THE OBJECTS HAVE NO "NAMEABLE" TYPE
- GIVEN SEVERAL ARRAY OBJECTS DECLARED IN A SINGLE ANONYMOUS ARRAY TYPE DECLARATION, EACH OBJECT IS CONSIDERED TO BE OF A <u>DIFFERENT</u> ANONYMOUS ARRAY TYPE.

EXAMPLE:

```
A, B : array (1 .. 3) of Boolean;
A := B;  --**ILLEGAL**--

type AB_Array_Type is array (1 .. 3) of Boolean;
A, B : AB_Array_Type;
A := B;  --**LEGAL**--
```

INSTRUCTOR NOTES

THIS FOIL PROVIDES THE MOTIVATION FOR THE NEXT FOIL.

CONSTRAINED ARRAY TYPES

- THE KIND OF ARRAY WE'VE BEEN DISCUSSING SO FAR IS A CONSTRAINED ARRAY, IN WHICH THE BOUNDS FOR THE INDEX ARE GIVEN IN THE TYPE DECLARATION

TYPE DECLARATIONS

```
type Temperature_Type is array (1 .. 30) of Float;

type Point_Characteristic_Type is (Object, Not_Object);

type Coordinate_3D_System is array (0 .. 239, 0 .. 319, 0 .. 239)
              of Point_Characteristic_Type;
```

OBJECT DECLARATIONS

```
Boiler_Room_Temperature : Temperature_Type;

Tank_Position           : Coordinate_3D_System;
```

INSTRUCTOR NOTES

THIS WAS RECOGNIZED AS ONE OF THE PROBLEMS WITH OTHER LANGUAGES (I.E., Pascal) AND THUS PROVIDED IN THE DEVELOPMENT OF THE Ada LANGUAGE.

RATIONALE FOR UNCONSTRAINED ARRAY TYPES

- ALLOWS SEVERAL ARRAY OBJECTS OF THE SAME TYPE BUT WITH DIFFERENT ARRAY BOUNDS

  PROBLEM:

  IN A MESSAGE-PASSING SYSTEM, EACH MESSAGE PACKET IS OF DIFFERENT LENGTHS.

  Ada SOLUTION:

  DECLARE THEM AS UNCONSTRAINED ARRAY TYPES WHOSE RANGE OF INDICES IS NOT DEFINED UNTIL AN ARRAY OBJECT IS DECLARED

INSTRUCTOR NOTES

- THE MAIN POINT IS THAT WE WANT A LIST OF 5 FREQUENCIES TO BE OF THE SAME TYPE AS A LIST OF 10 FREQUENCIES SO THAT WE CAN PERFORM THE SAME OPERATIONS ON THEM.

- CODE ON NEXT SLIDE

- AVIATION BACKGROUND INFORMATION:

  - UNICOM FREQUENCIES ARE USED AT PRIVATELY OPERATED ADVISORY STATIONS LOCATED AT MANY AIRPORTS. THE STATION PROVIDES GENERAL INFORMATION (WIND, RUNWAY IN USE, AIRCRAFT SERVICES, KNOWN TRAFFIC ...) THERE ARE 3 UNICOM FREQUENCIES AND EACH STATION IS ASSIGNED ONE OF THESE THREE.
  - THE HANSCOM AND LAWRENCE FREQUENCIES INCLUDE THE FREQUENCIES FOR CONTROL TOWER, GROUND CONTROL, EMERGENCY COMMUNICATIONS, MULTICON, VOR CHANNEL, AND ATIS.
  - VOR (VERY HIGH FREQUENCY OMNIDIRECTIONAL RANGE) TRANSMITS A RADIO SIGNAL WHICH CAN BE PICKED UP BY THE AIRCRAFT AND USED FOR NAVIGATION. THE FREQUENCIES LISTED ARE THOSE ON THE NORTHERN HALF OF THE NEW YORK SECTIONAL MAP, COVERING SOUTHERN ME, SOUTHERN NH, NORTHERN NY, AND MA.
  - ATIS (AIRPORT TERMINAL INFORMATION SERVICE) BROADCASTS CONTINUOUSLY AIRPORT INFORMATION SUCH AS RUNWAY NEWS, WEATHER CONDITIONS, AND SPECIAL ADVISORIES. THE FREQUENCIES LISTED ARE ATIS FOR FIVE MASSACHUSETTS AIRPORTS WHICH HAVE ATIS.

RATIONALE FOR UNCONSTRAINED ARRAY TYPES

- CONSIDER THE VHF AND UHF BANDS WHICH ARE USED IN CIVILIAN AND MILITARY AVIATION COMMUNICATIONS. FOR VARIOUS APPLICATIONS, WE NEED TO GROUP DIFFERENT FREQUENCIES. NOT ALL OF THE APPLICATIONS WILL GROUP THE SAME NUMBER OF FREQUENCIES. THE VARIOUS GROUPS MAY BE REPRESENTED BY ARRAYS OF THE SAME TYPE BUT WITH DIFFERENT LENGTHS .. UNCONSTRAINED ARRAY.

Unicom_List	Hanscom_Freq_List	NY_Sectional_North_VOR_List		Lawrence_Freq_List	MA_ATIS_List
122.7	118.5	115.2	113.0	112.5	124.6
122.8	121.5	117.1	115.1	120.0	135.0
123.0	121.7	116.5	117.8	121.5	125.1
	122.95	112.7	115.0	121.7	126.6
	124.6	114.4	110.2	122.8	123.8
		112.9	112.1		117.8
		113.7	116.8		
		112.4	112.6		
		109.4	108.6		
		110.6	109.8		
		114.5	117.0		
		117.4	115.2		
		114.3			
		111.8	108.4		

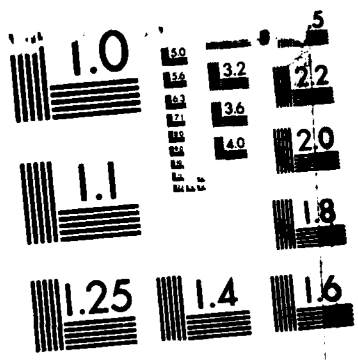
INSTRUCTOR NOTES

- STRESS THAT THE COMPONENT TYPE MUST BE A CONSTRAINED TYPE.
- <> IS ONE LEXICAL ELEMENT
- Matrix_1 AND Matrix_5 HAVE DIFFERENT INDEX BOUNDS IN THE FIRST DIMENSION, BUT THE SAME LENGTH. IT IS POSSIBLE TO ASSIGN Matrix_1 TO Matrix_5 OR VICE VERSA. COMPONENTS ARE ASSIGNED ACCORDING TO RELATIVE POSITION IN THE ARRAY, NOT THE INDEX VALUES NAMING THAT POSITION.
- AN ARRAY TYPE DECLARATION IS EITHER CONSTRAINED IN EVERY DIMENSION OR UNCONSTRAINED IN EVERY DIMENSION. THERE IS NO SUCH THING AS A PARTIALLY CONSTRAINED ARRAY TYPE.
- POINT OUT THAT TO DETERMINE THE ACTUAL BOUNDS ON AN OBJECT OF AN UNCONSTRAINED TYPE, WE CAN USE ATTRIBUTES SUCH AS 'FIRST AND 'LAST.
- POINT OUT THAT Vector_1 AND Vector_2 ARE OF THE SAME TYPE EVEN THOUGH THEIR ACTUAL INDEX BOUNDS APPEAR DIFFERENT (THE SAME GOES FOR Matrix_1 AND Matrix_5).
- IT IS POSSIBLE TO WRITE A SUBPROGRAM TO MANIPULATE ALL VALUES IN SOME UNCONSTRAINED ARRAY TYPE, REGARDLESS OF THEIR LENGTH. WITHOUT UNCONSTRAINED ARRAY TYPES, ALL ARRAYS TO BE MANIPULATED BY THE SUBPROGRAM WOULD HAVE TO BE THE SAME TYPE (WHICH IMPLIES THEY WOULD HAVE THE SAME INDICES).

AD-A166 366 ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA PROGRAMMING L202 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07-83-C-K514

7/8

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCONSTRAINED ARRAY TYPES

SYNTAX:

```
type Type_Name is array (Index_Subtype range<>{, Index_Subtype range<>})
                                            of Component_Type;
```

EXAMPLES:

TYPE DECLARATION

```
type Frequency_Type is delta 0.025 range 30.0 .. 3000.0;  -- cycles in MHz
type Frequency_List_Type is array (Positive range <>) of Frequency_Type;
type Matrix_Type is array (Integer range <>, Integer range <>) of Float;
```

OBJECT DECLARATION

```
Unicom_List                   : Frequency_List_Type (1 .. 3)  :=
   (122.7, 122.8, 123.0);
Hanscom_Freq_List             : Frequency_List_Type (1 .. 5)  :=
   (118.5, 121.5, 121.7, 122.95, 124.6);
NY_Sectional_North_VOR_List : Frequency_List_Type (1 .. 27) :=
   (115.2, 117.1, 116.5, 112.7, 114.4, 112.9, 113.7, 112.4, 109.4, 110.6, 114.5,
    117.4, 114.3, 111.8, 113.0, 115.1, 117.8, 115.0, 110.2, 112.1, 116.8, 112.6,
    108.6, 109.8, 117.0, 115.2, 108.4);
Lawrence_Freq_List            : Frequency_List_Type (1 .. 5)  :=
   (112.5, 120.0, 121.5, 121.7, 122.8);
MA_ATIS_List                  : Frequency_List_Type (1 .. 6)  :=
   (124.6, 135.0, 125.1, 126.6, 123.8, 117.8);
Matrix_1                      : Matrix_Type (-4 .. 0, 1 .. 5);
Matrix_5                      : Matrix_Type (1 .. 5, 1 .. 5);
```

INSTRUCTOR NOTES

UNCONSTRAINED ARRAY TYPES
NOTABLE POINTS

- BOUNDS MUST BE SUPPLIED AT OBJECT DECLARATION TIME WITH AN <u>INDEX CONSTRAINT</u>. UNCONSTRAINED ARRAY TYPES LEAVE THE SPECIFICATION OF SPECIFIC BOUNDS TO THE USE OF THE TYPE IN AN OBJECT DECLARATION.

  EXAMPLE:

```
type Unconstrained_Vector_Type is array (Integer range <>) of Float;
Vector_1 : Unconstrained_Vector_Type (1 .. 5);
```

- DIFFERENT OBJECTS OF THE TYPE CAN HAVE DIFFERENT BOUNDS

  EXAMPLE:

```
type Matrix_Type is array (Integer range <>, Integer range <>) of Float;
Matrix_1 : Matrix_Type (-4 .. 0, 1 .. 5);
Matrix_5 : Matrix_Type (1 .. 5, 1 .. 5);
```

- BOUNDS FOR ARRAY TYPE DECLARATIONS MUST BE EITHER ALL CONSTRAINED OR ALL UNCONSTRAINED.

```
type Not_Allowed_Type is array
     (I .. 5, Integer range <>) of Integer;    --**ILLEGAL
```

INSTRUCTOR NOTES

THE LAST ELEMENT FOR ANY COLUMN IS THE SCALE FACTOR FOR THAT COLUMN THUS

$\begin{bmatrix} 3 \\ 4 \\ 5 \\ 1 \end{bmatrix}$ IS EQUIVALENT TO $\begin{bmatrix} 6 \\ 8 \\ 10 \\ 2 \end{bmatrix}$ BUT IS NOT EQUIVALENT TO $\begin{bmatrix} 6 \\ 8 \\ 10 \\ 1 \end{bmatrix}$

MOTIVATION FOR SCALE FACTOR IS THAT MATRIX MANIPULATION (TRANSLATIONS, ROTATIONS, ETC.) MAY CHANGE THE SCALE FACTOR.

A VECTOR REPRESENTS A SINGLE POINT WHEREAS A MATRIX REPRESENTS A COLLECTION OF POINTS. WITHIN A MATRIX, A COLUMN REPRESENTS A POINT AND ALL POINTS ARE IN SOME COORDINATE SYSTEM. FOR EXAMPLE, THE FIGURE SHOWN IS REPRESENTED BY 6 POINTS, THE FIRST THREE NUMBERS IN EACH COLUMN REPRESENTING THE COORDINATES AND THE LAST BEING THE SCALE FACTOR.

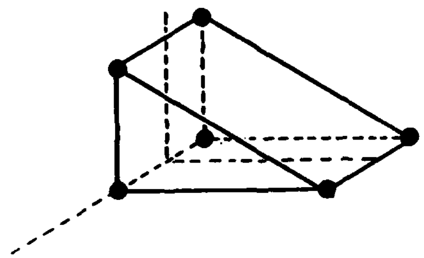
RATIONALE FOR UNCONSTRAINED ARRAY TYPE
ROBOTICS SYSTEM EXAMPLE

- IN A ROBOTICS SYSTEM, WE MAY WISH TO MANIPULATE AN OBJECT IN VARIOUS COORDINATE SYSTEMS.
- AN OBJECT IS REPRESENTED AS A MATRIX WHERE
  - THE NUMBER OF COLUMNS CORRESPOND TO THE NUMBER OF POINTS USED TO DESCRIBE THE OBJECT
  - THE NUMBER OF ROWS MODEL THE COORDINATE FRAME FIXED IN THE OBJECT PLUS A SCALE FACTOR

FOR EXAMPLE

$$\begin{bmatrix} 1.0 & -1.0 & -1.0 & 1.0 & 1.0 & -1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 4.0 & 4.0 \\ 0.0 & 0.0 & 2.0 & 2.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix} \qquad \begin{bmatrix} 1.0 & -1.0 & 1.0 & -1.0 \\ 0.0 & 1.0 & 1.0 & 0.0 \\ 2.0 & 2.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 1.0 \end{bmatrix}$$

6 POINTS   4 POINTS



```
type Object_Matrix_Type is array (Natural range <>, Natural range <>) of Float;
```

INSTRUCTOR NOTES

- POINT OUT THAT THE PROCEDURE'S SOURCE CODE DOESN'T CHANGE AS A RESULT OF THE TYPE BEING DEFINED AS AN UNCONSTRAINED ARRAY. IMPLICATION: THIS PROCEDURE WORKS REGARDLESS OF THE SIZE OF THE ARRAY.
- POINT OUT THAT THE FORMAL PARAMETER WILL INHERIT THE INDEX CONSTRAINT OF THE ACTUAL. THIS IS ACTUALLY THE MOST IMPORTANT REASON FOR UNCONSTRAINED ARRAY TYPES. THEY CAN BE USED AS THE PARAMETER TYPE IN A SUBPROGRAM PARAMETER LIST. THE ACTUAL BOUNDS ARE SUPPLIED BY THE ACTUAL PARAMETERS WHEN THE SUBPROGRAM IS CALLED.
- INSTRUCTORS SHOULD EXPLAIN THAT Rotation_Matrix MUST BE CONSTRAINED. THIS IS DONE THROUGH THE USE OF THE RANGE ATTRIBUTE AND PROVIDES A SQUARE MATRIX OF THE APPROPRIATE SIZE. A ROTATION MATRIX IS GENERALLY DEFINED TO BE A SQUARE MATRIX. THUS A 4 x 4 ROTATION TIMES A 4 x 6 SET OF POINTS YIELDS A 4 x 6 SET OF POINTS.
- FOR INTERESTED STUDENTS, IN THREE DIMENSIONS, THE ROTATION MATRIX ABOUT THE X_AXIS IS GIVEN BY:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\emptyset) & -\sin(\emptyset) & 0 \\ 0 & \sin(\emptyset) & \cos(\emptyset) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

WHERE Ø IS THE ANGLE OF ROTATION.

RATIONALE FOR UNCONSTRAINED ARRAY TYPE
ROBOTICS SYSTEM EXAMPLE

WE CAN NOW DECLARE A SINGLE PROCEDURE THAT WILL PERFORM THE SAME OPERATION ON MATRICES OF ANY SIZE.

CONTEXT:

```
type Object_Matrix_Type is array (Natural range <>, Natural range <>) of Float;
type Axis_Type is (X, Y, Z);
--     ASSUME MATRIX MULTIPLICATION IS DEFINED
--     ASSUME TRIG FUNCTIONS ARE AVAILABLE
```

EXAMPLE:

```
procedure Rotate_Object (Object_Matrix : in out Object_Matrix_Type;
                               Degrees : in Float;
                                  Axis : in Axis_Type) is
      Rotation_Matrix : Object_Matrix_Type (Object_Matrix'Range(1),
                             Object_Matrix'Range(1));
begin
      case axis is
            when X => Rotation_Matrix := ...
            when Y => ...
            when Z => ...
      end case;
      Object_Matrix := Multiply (Rotation_Matrix, Object_Matrix);
end Rotate_Object;
```

INSTRUCTOR NOTES

CONSTRAINED ARRAY SUBTYPES

- IT IS POSSIBLE TO DEFINE A SUBTYPE OF AN UNCONSTRAINED ARRAY TYPE, CONSISTING OF ALL VALUES IN THE TYPE HAVING A GIVEN INDEX CONSTRAINT.

```
type Unconstrained_Vector_Type is array (Integer range <>) of Float;
subtype Vector_of_10_Subtype is Unconstrained_Vector_Type (1 .. 10);
```

- THE FOLLOWING VARIABLE DECLARATIONS ARE EQUIVALENT:

```
Vector_1 : Unconstrained_Vector_Type (1 .. 10);
Vector_1 : Vector_of_10_Subtype;
```

ASK THE CLASS THE FOLLOWING QUESTION:

GIVEN A STRING VARIABLE S, HOW DO YOU FIND OUT HOW MANY CHARACTERS ARE IN IT?

ANSWER: S'Length

THE TYPE STRING

- PREDEFINED UNCONSTRAINED ONE DIMENSIONAL ARRAY TYPE WITH CHARACTER COMPONENTS AND INTEGER INDICES

- DEFINITIONS:

```
subtype Positive is Integer range 1 .. Integer'Last;
type String is array (Positive range <>) of Character;
```

- LOWER BOUND OF OBJECT INDICES MUST BE 1 OR GREATER

- POSSIBLE PROGRAMMER-SUPPLIED DEFINITIONS:

```
subtype Name_String is String (1 .. 8);
type Name_List_Type is array (1 .. 5) of String (1 .. 8);
```

NOTE: A CONSTRAINT IS IMPOSED ON THE ARRAY COMPONENT TYPE, STRING OBJECT DECLARATIONS

```
Name           : String (1 .. 15);
Seminar_Roster : Name_List_Type;
```

INSTRUCTOR NOTES

POINT OUT THAT ALL THE ENTITIES LISTED ARE STRING LITERALS. (REVIEW FROM SECTION 3).

BECAUSE OF THE DEFINITION OF THE ASCII CHARACTER SET, ALL UPPER CASE LETTERS ARE PRESUMED TO PRECEDE ALL LOWER CASE LETTERS WHEN COMPARING STRINGS.

NOTICE THAT THE STRING "10" IS LESS THAN "2".

REVIEW RELATIONAL ARRAY TYPE OPERATIONS

- THE ARRAYS DO NOT NEED TO BE OF THE SAME LENGTH

```
"A" < "AA"          -- FOLLOWS DICTIONARY ORDERING
"A" < "abc"         -- CAPITALS PRECEDE LOWER CASE
```

ASK STUDENTS WHY THE ILLEGAL EXAMPLE IS ILLEGAL. (THEY ARE OF DIFFERENT TYPES, 'C' IS A CHARACTER LITERAL WHILE "C" IS A STRING LITERAL.)

STRING TYPE OPERATIONS

RELATIONAL OPERATIONS

LEXICAL ORDERING (ALMOST ALPHABETICAL)

```
"A" < "a"            --  true
"abc" < "acc"        --  true
"A" < "AAA"          --  true
"AZURE" < "BLUE"     --  true
'C' < "C"            --  **illegal;
"ZETA" < "alpha"     --  true
"10" < "z"           --  true
```

EQUALITY/INEQUALITY

```
"BITS" /= "BIT"      --  true
"BITS" =  "BYTE"     --  false
```

INSTRUCTOR NOTES

POINT OUT THAT THE String VALUES IN Seminar_Roster MUST BE BLANK-FILLED TO ENSURE 8 CHARACTERS.

STRING TYPE OPERATIONS

- ASSIGNMENT

  CONTEXT:

```
type Name_List_Type is array (1 .. 5) of String (1 .. 8);

Seminar_Roster : Name_List_Type;
```

  EXAMPLE:

```
Seminar_Roster := ("MARGARET",
                   "JONATHAN",
                   "ANTHONY ",
                   "MARY ANN",
                   "JANE    ");
```

- CATENATION

  EXAMPLE:

```
Game : String(1 .. 13) := "CAT " & "AND " & "MOUSE";
```

- ATTRIBUTES

  - SAME AS ALL ARRAY TYPES (S'FIRST, S'LAST, S'RANGE, S'LENGTH)

INSTRUCTOR NOTES

STRING TYPE
NOTABLE POINTS

- BE CAREFUL IN ASSIGNING VALUES TO STRINGS

CONTEXT:

```
     Item : String (1 .. 5);
```

EXAMPLES:

```
     1.  Item(1)      := "(";          -- **ILLEGAL**
                                       -- CANNOT ASSIGN A STRING VALUE TO
                                             A CHARACTER COMPONENT
     2.  Item(1)      := '(';          -- LEGAL
                                       -- ASSIGN A CHARACTER TO A CHARACTER
                                             COMPONENT
     3.  Item(1 .. 1) := "(";          -- LEGAL
                                       -- ASSIGN A STRING VALUE TO A SLICE
     4.  Item         := "BIT";        -- **ILLEGAL**
                                       -- THE STRING WAS DECLARED OF LENGTH 5 AND
                                             BIT ONLY HAS LENGTH OF 3
     5.  Item         := "BITbb";      -- LEGAL
                                       -- BLANK FILLED
     6.  Item(1 ..3)  := "BIT";        -- LEGAL
                                       -- ASSIGNED TO A SLICE
```

INSTRUCTOR NOTES

THIS IS AN IMPORTANT POINT. TEXT I/O FOR ARRAYS IS DONE ON A COMPONENT BY COMPONENT BASIS.

FOR INSTRUCTORS INFORMATION. I/O FOR WHOLE ARRAYS IS PROVIDED BY Direct_IO AND Sequential_IO. MAY WANT TO MENTION THIS TO THE STUDENTS. REFER TO SECTION 15 WHERE THESE IO PACKAGES WILL BE COVERED.

I/O FOR ARRAY TYPES

- NONE IS PREDEFINED (EXCEPT FOR String)

- MUST PERFORM I/O ON A COMPONENT BY COMPONENT BASIS IF USING Text_IO

GO OVER THE TUTORIAL IN THE LAB MANUAL ON I/O WITH USER FILES, AND THEN ASSIGN EXERCISES 13 THROUGH 21. THESE EXERCISES USE I/O WITH USER FILES SO IT IS IMPORTANT TO GO OVER IT. THE STUDENTS MAY NOT BE ABLE TO COMPLETE THEM ALL. STRESS THAT THEY AT LEAST DO NUMBER 18.

ASSIGN CHAPTER 8 OF THE PRIMER.

MENTION THAT THE use CLAUSE IS OPTIONAL. MANY COMPANIES' RECOMMENDED CODING PRACTICE SUGGEST NOT TO INCLUDE THE use CLAUSE. IT INCREASES THE UNDERSTANDABILITY AND THEREFORE THE MAINTENANCE OF THE PROGRAM TO NOT INCLUDE A use CLAUSE.

EXPLAIN THAT <u>Get</u> DETERMINES THE LENGTH OF THE STRING AND ATTEMPTS THAT NUMBER OF Get OPERATIONS FOR SUCCESSIVE CHARACTERS OF THE STRING. NO OPERATION IS PERFORMED IF THE STRING IS NULL. <u>Get_Line</u> REPLACES SUCCESSIVE CHARACTERS OF THE SPECIFIED STRING BY SUCCESSIVE CHARACTERS READ FROM THE SPECIFIED INPUT FILE. READING STOPS IF THE END OF LINE OR END OF STRING IS MET. CHARACTERS NOT REPLACED ARE LEFT UNDEFINED. N CONTAINS THE INDEX VALUE OF THE LAST CHARACTER REPLACED IN S. <u>Put</u> DETERMINES THE LENGTH OF THE GIVEN STRING AND ATTEMPTS THAT NUMBER OF PUT OPERATIONS FOR SUCCESSIVE CHARACTERS OF THE STRING. <u>Put_Line</u> CALLS THE PUT PROCEDURE FOR THE GIVEN STRING AND ADVANCES ONE LINE.

I/O FOR THE TYPE STRING

- WRITE AT TOP OF COMPILATION UNIT:

```
with Text_IO; use Text_IO;
```

- TO PERFORM INPUT AND OUTPUT:

  CONTEXT:

```
S : String (1 .. 80);
N : Natural;
```

  EXAMPLE:

```
Get (S);
Put (S);

Put_Line (S);
Get_Line (S, N);
```

INSTRUCTOR NOTES

EACH OF THESE FORMS WILL BE ADDRESSED IN DETAIL.

CONTROL STRUCTURE-LOOPS
THREE BASIC FORMS

1. SIMPLE LOOP
   - REPEATEDLY EXECUTES STATEMENTS AD INFINITUM

2. FOR LOOP
   - REPEATEDLY EXECUTES STATEMENTS FOR SPECIFIC VALUES OF THE LOOP PARAMETER

3. WHILE LOOP
   - REPEATEDLY EXECUTES STATEMENTS WHILE SOME CONDITION IS TRUE

INSTRUCTOR NOTES

THIS AND THE NEXT FOIL ADDRESS THE SIMPLE INFINITE LOOP.

POINT OUT THAT THE LOOP WILL EXECUTE AD INFINITUM UNLESS ONE OF THE STATEMENTS IN THE LOOP IS AN EXIT STATEMENT, RETURN STATEMENT, OR A goto STATEMENT WHICH PASSES CONTROL TO OUTSIDE THE LOOP.

FIRST FORM - SIMPLE LOOP

SYNTAX:

```
[Loop_Identifier:]
      loop
         -- Statements
      end loop [Loop_Identifier];
```

- LOOP EXECUTES AD INFINITUM

INSTRUCTOR NOTES

POINT OUT THAT LOOP NAME IS OPTIONAL.

POINT OUT THAT THE LOOP NAME MUST BE GIVEN AT BOTH THE TOP AND BOTTOM OF THE LOOP OR IN NEITHER PLACE.

THE INSTRUCTOR SHOULD EXPLAIN THAT Put_Horizontal DISPLAYS 320 CHARACTERS ACROSS STARTING AT THE PIXEL POSITION GIVEN IN THE AGGREGATE. LIKEWISE, Put_Vertical DISPLAYS 240 CHARACTERS DOWN STARTING AT THE POSITION GIVEN BY THE AGGREGATE. RATHER THAN DECLARING A NEW TYPE FOR THE VERTICAL BORDER WE USE AN APPROPRIATE SLICE TO DISPLAY THE IDENTICAL PATTERN ($).

CONTEXT

```
type Point_Type is array (1 .. 2) of Natural;
procedure Put_Horizontal (S : in String; Left_Corner : Point_Type);
procedure Put_Vertical   (S : in String; Top_Corner  : Point_Type);
```

SIMPLE LOOP - EXAMPLE

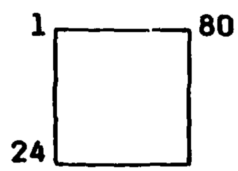
- STATEMENTS REPEATED AD INFINITUM

- LOOP MAY BE GIVEN A LABEL, CALLED A LOOP IDENTIFIER

- THE LOOP IDENTIFIER MUST BE REPEATED AFTER THE END LOOP

EXAMPLE:

```
procedure Display_Screen_Border is
   type Screen_Line_Type is array ( 1 .. 80) of Character;
   Border : Screen_Line_Type := (others => '$');

begin -- Display_Screen_Border
   Refresh:                                          -- loop identifier
      loop
         Put_Horizontal (Border,(0, 0));
         Put_Vertical (Border (1 .. 24), (0, 0));
         Put_Horizontal (Border,(23, 0));
         Put_Vertical (Border(1 .. 24), (0, 79));
      end loop Refresh;                              -- loop identifier repeated
end Display_Screen_Border;
```



INSTRUCTOR NOTES

REVIEW ATTRIBUTES : List'Range IS EQUIVALENT TO List'First .. List'Last

REMIND STUDENTS THAT A DISCRETE RANGE (Integer OR Enumeration) CAN BE EITHER:

- Subtype_Name
- Lower_Bound .. Upper_Bound
- Subtype_Name range Lower_Bound .. Upper_Bound

HERE, THE SECOND FORM. POINT OUT THAT reverse IS OPTIONAL. RANGE IN REVERSE IS ALWAYS WRITTEN IN ASCENDING ORDER. THE LOOP PARAMETER ASSUMES VALUES FROM THE LATTER VALUE <u>DOWN TO</u> THE FIRST VALUE.

SECOND FORM - FOR LOOP

SYNTAX:

```
        [Loop_Identifier:]
           for Loop_Parameter in [reverse] Discrete Range
              loop
                 -- statements
               end loop [Loop_Identifier];
```

- STATEMENTS EXECUTED ONCE FOR EACH VALUE IN RANGE
- LOOP PARAMETER INCREMENTED IN STEPS OF 1 AUTOMATICALLY (OR -1 IF reverse IS USED)
- LOOP PARAMETER IS LOCAL TO THE LOOP
  - NOT AVAILABLE OUTSIDE THE LOOP
  - NOT DECLARED IN OBJECT DECLARATION

INSTRUCTOR NOTES

WE ARE ATTEMPTING TO GET ACROSS THE MAIN POINTS CONCERNING for LOOPS.

THIS SLIDE'S INTENT IS TO COVER:

- INCREMENTING - THE EXAMPLE SHOWS HOW TO ACCESS EVERY SECOND ROW
- DISCRETE RANGE
- INDEX IN VARIANT
- SCOPE

SPEND SOME TIME ON THIS SLIDE GOING OVER THE NOTABLE POINTS.

POINT THAT for LOOPS CAN BE NESTED TO ANY DEPTH. A NESTED for LOOP IS COMMONLY USED IN ITERATING THROUGH THE ELEMENTS OF A MULTIDIMENSIONAL MATRIX.

WALK THROUGH CODE, POINTING OUT:

- IF Lower_Bound > Upper_Bound LOOP NOT EXECUTED
- IF Lower_Bound = Upper_Bound LOOP EXECUTED ONCE
- IF Lower_Bound < Upper_Bound LOOP EXECUTED FOR VALUES
  IN Lower_Bound TO Upper_Bound IN STEPS OF 1

EXAMPLES OF for LOOPS

FILL THE GRAPHICS SCREEN SUCH THAT IT ALTERNATES ROWS OF WHITE AND BLACK PIXELS:

CONTEXT:

```
type Color_Component_Type   is (Red, Green, Blue);
subtype Color_Quantity_Type is Integer range 0 .. 15;
type Color_Type             is array (Color_Component_Type) of
                                        Color_Quantity_Type;
type Color_Screen_Type     is array (0 ..239, 0 .. 319) of Color_Type;
Pattern  : Color_Type        := (Red => 15, Green => 15, Blue => 15);
Screen   : Color_Screen_Type := (others => (others => (Red => 0, Green => 0,
                                 Blue => 0)));
Dummy, J : Integer           := Screen'Last(1)/2;
```

EXAMPLE:

```
for I in Screen'First(1) .. Screen'Last(1)/2 loop
   I := I * 2;  -- **ILLEGAL
   J := I * 2;  --   LEGAL
   for K in Screen'Range(2) loop
      Screen (J, K) := Pattern;
   end loop;
end loop;
I := Dummy;          -- **ILLEGAL BECAUSE I IS NOT KNOWN
J := Dummy;          -- **LEGAL
```

NOTABLE POINTS:

- CANNOT MAKE ASSIGNMENTS TO LOOP PARAMETER I INSIDE THE LOOP.
- THE LOOP IS INCREMENTED IN STEPS OF 1. HOWEVER, BECAUSE OF THE ASSIGNMENT TO J THE EFFECT IS INCREMENTING IN STEPS OF 2.
- THE ONLY WAY TO SAVE THE VALUE OF THE INDEX (LOOP PARAMETER) FOR USE OUTSIDE THE LOOP IS TO ASSIGN IT TO SOME OTHER VARIABLE.

INSTRUCTOR NOTES

POINT OUT THAT THE CONDITION IS CHECKED AT THE TOP OF THE LOOP BEFORE EACH EXECUTION OF THE LOOP.

POINT OUT THAT THE LOOP IS TERMINATED WHEN THE CONDITION IS FALSE.

THIRD FORM - WHILE LOOP

SYNTAX:

```
[Loop_Identifier:]
   while Boolean_Expression
      loop
         -- Statements
      end loop [Loop_Identifier];
```

- STATEMENTS EXECUTED AS LONG AS Boolean_Expression IS True
- LOOP TERMINATED WHEN THE CONDITION IS False

INSTRUCTOR NOTES

ASSUMES THAT YOU CAN ALWAYS ADD ANOTHER 1/10TH OF A GALLON TO A FULL TANK.

WHILE LOOP - EXAMPLE

CONTEXT:

```
Gas_In_Tank : Float;
Tank_Size   : constant Float := 18.0;
```

EXAMPLE:

```
Fill_Tank:
   while Gas_In_Tank + 0.1 <= Tank_Size
      loop
         Pump_Gas;            -- adds 1/10 gallon
         Gas_In_Tank := Gas_in_Tank + 0.1;
      end loop Fill_Tank;
```

INSTRUCTOR NOTES

POINT OUT THAT WHEN A NAME OF A LOOP (LOOP IDENTIFIER) IS SPECIFIED IN THE EXIT STATEMENT THEN <u>THAT</u> LOOP IS EXITED. THIS IS PERTINENT FOR EXITING NESTED LEVELS OF LOOPS. SPECIFICALLY IF THREE LOOPS ARE NESTED, AND THE MIDDLE LOOP HAS A LOOP IDENTIFIER AND THE INNERMOST SEQUENCE OF STATEMENTS CONTAINS THE STATEMENT EXIT MIDDLE; THEN THE INNERMOST AND MIDDLE LOOP ARE EXITED.

POINT OUT DISTINCTION BETWEEN CONDITIONAL AND UNCONDITIONAL EXIT, AND THAT FOR CONDITIONAL EXIT, THE CONDITION EXPRESSION MUST BE Boolean.

GOOD STRUCTURED PROGRAMMING DICTATES THAT LOOPS SHOULD HAVE ONLY ONE ENTRY POINT AND ONE EXIT POINT. THE LOOP WITH MORE THAN ONE EXIT STATEMENT VIOLATES THE DOCTRINE OF STRUCTURED PROGRAMMING. IN MOST CASES A LOOP WITH A SINGLE EXIT CAN BE REWRITTEN AS EITHER A WHILE LOOP OR POSSIBLY A FOR LOOP. THESE FORMS SHOULD BE USED IF POSSIBLE.

EXIT STATEMENT WILL BE SEEN AGAIN IN CHAPTER 14 (EXCEPTIONS).

LOOP WITH EXIT

- EXIT STATEMENT MAY BE USED TO EXPLICITLY EXIT A LOOP

- EXIT TAKES VARIOUS FORMS

```
exit;                                       -- unconditional exit
exit when Boolean_Expression;               -- conditional exit
exit Loop_Name;                             -- unconditional exit
exit Loop_Name when Boolean_Expression;     -- conditional exit
```

INSTRUCTOR NOTES

"WHEN" IS OPTIONAL ON EXIT STATEMENTS. IT SPECIFIES A CONDITION WHICH NEEDS TO BE MET BEFORE EXITING. WHEN IT IS NOT SPECIFIED YOU HAVE AN UNCONDITIONAL EXIT.

POINT OUT AGAIN THAT THIS EXAMPLE COULD BE WRITTEN AS A LOOP WITHOUT AN EXIT. ASK THE STUDENTS TO SUGGEST HOW TO WRITE IT.

ANSWER:

```
Get (Command);
while Command /= Halt loop
   Process (Command);
   Get (Command);
end loop;
```

USE OF THE exit STATEMENT

IF AN exit STATEMENT IS ENCOUNTERED INSIDE A LOOP, THE LOOP TERMINATES AND CONTROL PASSES TO THE POINT IMMEDIATELY AFTER THE APPROPRIATE "end loop"

CONTEXT:

```
type Command_Type is (About_Face, Right, Left, Forward_March, Halt);
Command : Command_Type
```

EXAMPLE:

```
loop
   Get (Command);              -- procedure call
exit when Command = Halt;      -- conditional exit
   Process (Command);          -- procedure call
end loop;
```

EXERCISE: REWRITE THIS LOOP WITHOUT THE exit STATEMENT.

INSTRUCTOR NOTES

THIS EXAMPLE ILLUSTRATES THAT AN EXIT STATEMENT CAN OCCUR INSIDE SEVERAL LEVELS OF NESTED LOOPS.

THE NOTE IS A STRUCTURED PROGRAMMING TIP. WE SHOW THIS EXAMPLE ONLY TO ILLUSTRATE THAT IT CAN BE DONE. IT IS NOT RECOMMENDED PROGRAMMING PRACTICE.

POINT OUT AGAIN THAT LOOP PARAMETER VALUES MUST BE SAVED IN ORDINARY VARIABLES (Row AND Col) TO MAKE THEM AVAILABLE OUTSIDE THE LOOP.

THIS WOULD BE BETTER WRITTEN WITH ATTRIBUTES. ASK STUDENTS HOW ...

ANSWER:

```
for I in Matrix'Range(1)
...
for J in Matrix'Range(2)
```

FOR HISTORICAL INFORMATION Matrix_Type DECLARATION NEEDS "INTEGER RANGE" BECAUSE THE IMPLICIT CONVERSION OF -5 .. 10 TO THE PREDEFINED TYPE INTEGER CAN NOT BE ASSUMED.

REFER TO Ada REFERENCE MANUAL 3.6.1

LOOP WITH EXIT

CONTEXT:

```
type Matrix_Type is array (Integer range -5 .. 10, 1 .. 30) of Float;
Matrix      : Matrix_Type;
Row, Column : Integer := -99;
```

EXAMPLE:

```
Find:                                                -- loop identifier is Find
   for I in Integer range -5 .. 10
   loop
      for J in 1 .. 30
      loop
         if Matrix(I, J) = 0.0 then
            Row := I;
            Col := J;                                -- unconditional exit
      exit Find;
         end if;
      end loop;
   end loop Find;
```

NOTE: AS A GENERAL PROGRAMMING TIP, THE exit STATEMENT SHOULD ONLY BE USED TO EXIT THE INNERMOST LOOP STATEMENT THAT CONTAINS THE exit STATEMENT

INSTRUCTOR NOTES

"THERE ARE RULES SPECIFYING WHERE goto CAN AND CANNOT BRANCH TO. THEY ARE NOT COVERED HERE. WE DON'T RECOMMEND YOU USE THEM."

THE LABEL IN THE goto STATEMENT DOES NOT HAVE <<>> s. POINT OUT THAT THE REASON LABELS IN Ada HAVE THE DOUBLE BRACKET ENCLOSING IT IS BECAUSE Ada WANTS THESE ENTITIES TO CLEARLY STAND OUT IN THE CODE SO REVIEWERS AND MANAGERS WILL BE ABLE TO EASILY SPOT THEM IN THE CODE AND QUESTION THEIR USE. THE REASON Ada HAS THEM IN THE LANGUAGE AT ALL IS BECAUSE IT RECOGNIZES THAT SOME APPLICATION REQUIRE THEIR USE. (THE CODE WITHOUT goto WOULD BE OBSCURE AND UNMAINTAINABLE.) SUGGESTIONS FOR WHERE IT MAY BE APPROPRIATE TO USE THE goto ARE DISCUSSED IN CASE STUDIES 2.4.1 (THE USE OF EXCEPTIONS) AND 2.5.3 (REDUCING DEPTH OF NESTING) OF <u>Ada CASE STUDIES II</u> RESPECTIVELY.

THERE IS NO COLON FOLLOWING THE LABEL.

goto STATEMENT AND LABELS

- A LABEL IS AN IDENTIFIER ENCLOSED IN DOUBLE ANGLED BRACKETS:

  ```
  <<The_Devil>>
  ```

- ANY STATEMENT MAY BE LABELED

- THE LABEL APPEARS BEFORE THE FIRST WORD IN THE STATEMENT

- A goto STATEMENT TAKES THE FORM OF THE RESERVED WORD goto FOLLOWED BY THE LABEL IDENTIFIER AND SEMICOLON

  ```
  goto The_Devil;
  ```

- THERE ARE RESTRICTIONS ON WHERE goto'S MAY BRANCH

INSTRUCTOR NOTES

POINT OUT ALSO THAT goto STATEMENTS MAKE PROGRAM LOGIC DIFFICULT TO FOLLOW AND OPTIMIZATION DIFFICULT TO PERFORM.

goto STATEMENT

- IT'S THERE "JUST IN CASE"

- GOOD Ada PROGRAMMERS WON'T USE IT

- AN INDEPTH DISCUSSION ON LOOP STATEMENTS CAN BE FOUND IN THE REPORT Ada CASE STUDIES II SECTION 2.3.3.

POSSIBLE ANSWERS

WHILE LOOP:

```
function Search (Memory : in Memory_Type; Control_Sequence : in
   Bit_Sequence_Type) return Natural is
   Memory_Index : Natural := 256;

begin         -- Search
   Memory_Index := Memory'First;
   while Memory_Index <= Memory'Last and then Memory (Memory_Index)
            /= Control_Sequence loop
      Memory_Index := Memory_Index + 1;   -- BE SURE TO POINT OUT USE OF
   end loop;                              -- SHORT CIRCUIT CONTROL FORM
   return Memory_Index;
end Search;
```

FOR LOOP:

```
function Search (Memory : in Memory_Type; Control_Sequence : in
   Bit_Sequence_Type) return Natural is
   Memory_Index : Natural := 256;

begin         -- Search
   for Index in Memory'Range loop
      Memory_Index := Index;
      exit when Memory (Memory_Index) = Control_Sequence;
   end loop;
   if Memory (Memory_Index) /= Control_Sequence then Memory_Index := 256;
   end if;
   return Memory_Index;
end Search;
```

IN-CLASS EXERCISE

CONTEXT:

```
type Bit_Sequence_Type is array (0 .. 7) of Boolean;

type Memory_Type is array (0 .. 255) of Bit_Sequence_Type;
```

COMPLETE THE CODE:

WHERE FUNCTION Search SEARCHES THROUGH Memory FOR THE PARTICULAR Control_Sequence GIVEN AND RETURNS THE INDEX MATCHING Control_Sequence. THE Memory_Index RETURNED SHOULD HAVE THE VALUE OF 256 IF NO MATCHING Control_Sequence IS FOUND.

```
function Search (Memory : in Memory_Type; Control_Sequence : in Bit_Sequence_Type)
                 return Natural is
   Memory_Index : Natural := 256;
begin




   return Memory_Index;
end Search;
```

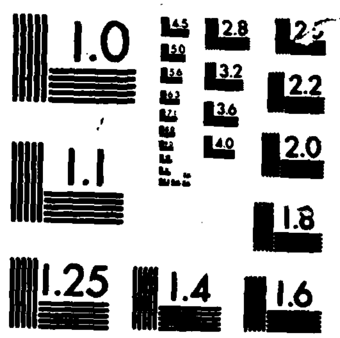
END

DTIC

5-86

AD-A166 366 ADA (TRADE NAME) TRAINING CURRICULUM BASIC ADA PROGRAMMING L202 TEACHER'S GUIDE VOLUME 1(U) SOFTECH INC WALTHAM MA 1986 DAAB07783-C-K514 8/8

UNCLASSIFIED F/G 9/2 NL



MICROCOP CHART

SUPPLEMENTARY

INFORMATION

AD-A166366

Material: Basic Ada Programming (L202), Volume I [redacted]

We would appreciate your comments on this material and would like you to complete this brief questionaire. The completed questionaire should be forwarded to the address on the back of this page. Thank you in advance for your time and effort.

1. Your name, company or affiliation, address and phone number.

2. Was the material accurate and technically correct?

   Yes ☐ No ☐

   Comments:

3. Were there any typographical errors?

   Yes ☐ No ☐

   If yes, on what pages?

4. Was the material organized and presented appropriately for your applications?

   Yes ☐ No ☐

   Comments:

5. General Comments:

END

DTIC

6-86